THOMSON

Premier 游戏开发丛书

含CD

游戏音效编程

[美] Mason McCuskey 著
华庆生 主译

cqup
•com
•cn

重庆大学出版社

用声音效果为你的游戏
——带来新的亮点

无论游戏有多么扣人心弦，无论可视化效果有多么逼真，如果缺少了背景音效，自然会显得美中不足。现在，技术上的新发展使你能够在声音方面做一些以前无法完成的事情，但是这也需要有一个向导来带你走过技术上的困难。《游戏声效编程》一书涉及到创建游戏声效和音乐时所遇到的问题，并提供了解决这些问题的方法。书中介绍了如何合成动态音乐和3D音效，在游戏中如何运用DirectPlay Voice实现实时聊天。如果你了解C++和DirectX，请随《游戏音效编程》进入令人激动的声音世界吧。

Mason McCuskey是一位有8年编程经历的软件工程师，他于1990年成功创立了自己的游戏公司Spin Studios。2000年，在游戏开发者协会第二届游戏程序竞赛中，Spin Studios公司入围前9名进入决赛。Mason也是Premier出版的《Special Effects Game Programming with DirectX》一书的作者，他还为gamedev.net和其他Web网站写了许多文章

Andre LaMothe是Xtreme Games公司的CEO，涉足计算机行业已经25年。他的工作经历涉及到2D/3D图形、人工智能、编译器设计、机器人技术、虚拟现实，以及电信学。他的书是游戏编程类书中最畅销的书籍，他的经验与指导体现于Premier出版的《游戏开发》丛书中

CD中所包含的内容

微软® DirectX® 9.0 SDK

BASS
SOUND SYSTEM v1.8

FMOD MUSIC AND SOUND
EFFECTS SYSTEM v3.61

OGG VORBIS SDK v1

ACID® XPRESS 演示版

SOUND FORGE® 演示版

包含微软许可软件，受“拆封即为接受”的最终用户许可协议约束。

ISBN 7-5624-3348-8
9 787562 433484 >

course
•com

ISBN 7-5624-3348-8
定价：42.80元

游戏音效编程

原著：［美］Mason McCuskey
主译：朱庆生
译者：王勇　陈波　葛亮

重庆大学出版社

Mason McCuskey
Beginning Game Audio Programming
ISBN: 1-59200-029-0

Copyright © 2003 by Premier Press, a division of Thomson Learning.

Original language published by Thomson Learning. All Rights reserved.
本书原版由汤姆森学习出版集团出版。版权所有,盗印必究。

Chongqing University Press is authorized by Thomson Learning to publish and distribute exclusively this simplified Chinese edition. This edition is authorized for sale in the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.
本书中文简体字翻译版由汤姆森学习出版集团授权重庆大学出版社独家出版发行。此版本仅限在中华人民共和国境内(不包括中国香港、澳门特别行政区及中国台湾)销售。未经授权的本书出口将被视为违反版权法的行为。
未经出版者预先书面许可,不得以任何方式复制或发行本书的任何部分。
981-265-254-x

版贸核渝字(2004)第34号

图书在版编目(CIP)数据

游戏音效编程/(美)麦卡斯基(McCuskey,M.)著;
朱庆生译. —重庆:重庆大学出版社,2005.3
(Premier 游戏开发丛书)
ISBN 7-5624-3348-8

Ⅰ.游... Ⅱ.①麦...②朱... Ⅲ.游戏—声音质量—程序设计 Ⅳ.TP311.5

中国版本图书馆 CIP 数据核字(2005)第016988号

游戏音效编程
Youxi Yinxiao Biancheng
[美]Mason McCuskey(麦卡斯基)著　　朱庆生　主译

出 版 者:重庆大学出版社　　社　　址:重庆市沙坪坝正街174号重庆大学(A区)内
网　　址:http://www.cqup.com.cn　　邮　　编:400030
电　　话:(023)65102378　65105781　　传　　真:(023)65103686　65105565
出 版 人:张鸽盛
责任编辑:王　斌　武雨南川　　版式设计:王　斌
责任校对:李定群　　责任印制:秦　梅
印 刷 者:自贡新华印刷厂印刷
发 行 者:全国新华书店经销
开　　本:787×960　1/16　　印张:20.5　　字数:411千
版　　次:2005年5月第1版　　2005年5月第1次印刷
书　　号:ISBN 7-5624-3348-8
印　　数:1—3 000
定　　价:42.80元

译者序

游戏音效编程在游戏开发中至关重要，它直接影响游戏的整体效果。本书作者从事游戏编程工作多年，在游戏音效编程方面具有丰富的实践经验。在本书中，他从音频基本知识入手，系统地介绍如何在游戏中添加声音或背景音乐等实用技术。全书分为两部分，第一部分“音频引擎基础”，重点介绍如何把 WAV 音频、MIDI 音频、MP3 音频、CD 音频和轨迹音乐等不同格式的音频文件引入游戏软件中，主要以 DirectMusic、DirectSound、DirectAudio 为开发工具，对每一种音频文件的使用方法进行详细描述，其目的是引导读者掌握播放和控制游戏音效的基本知识和技巧；第二部分“高级音频引擎”，重点介绍如何把动态音乐、3D 音乐、音频特效和环境音效等技术引入游戏软件中，主要以 Directx Audio、OPENAL、DirectPlay 等为软件工具，对多种高级音效技术逐一进行讲述，其目的是引导读者掌握动态音乐的基本术语、音频脚本的基本知识、音频可视化技术和高级游戏音效的实用编程技巧。本书配有大量源程序对所涉及的术语和基础知识进行讲解和支撑，并提供了包含实例源代码的配套光盘。

本书结构清晰，语言简洁，深入浅出，实例丰富，可操作性强，适用于游戏开发人员及广大游戏编程爱好者，也可作为高等院校相关专业师生的参考书籍。如果你正在进行游戏编程或希望将来开发游戏软件，为游戏添加音效是必不可少的，那么，选择《游戏音效编程》必有收获。

本书由朱庆生负责组织翻译和审校，参与翻译工作的人员还有王勇、陈波、葛亮。由于译者经验和水平有限，译文难免有不妥之处，欢迎广大读者批评指正。

译　者

2004 年 8 月于重庆

丛书编辑致读者的信

如果读者喜欢我,那么很可能也会喜欢游戏中的音乐和声效。过去在游戏开发过程中,音乐和声效并不是人们关注的要点,通常的做法是先罗列出所需要的声效和音乐段的清单,并确定播放质量标准,如16-bit 立体声、44 kHz 采样频率,然后雇某人去完成。这主要是因为开发者不想花费太多的时间去处理游戏音效。如在游戏中,可能会为飞机使用一些3D 音频,为游戏某处添加多普勒频移(doppler frequency shifting),或者为隧道中的声音改变音调等。但现在如果还这样做,就犯了一个大错。音频技术在过去的5~7年中以不可思议的速度发展并成熟。Windows 平台上 DirectAudio 的音频处理能力、硬件供应商在3D 声音建模方面的技术进步、丰富的超高频有损或无损音频压缩技术为游戏设计人员创建高质量游戏音效环境提供了极大的支持和方便。

一定要记住"声音比视频更重要"。事实上,声音可以创造紧张、兴奋、愤怒等效果。试着在没有声音的情况下玩你最喜欢的游戏,它还有吸引力吗? 这并不神秘,声音对你的大脑具有直接和立即的影响,它极大程度地抚慰或者刺激你的大脑。在任何情况下,都不能忽视声音。3D 图形的谜团已经解开,是时间了解声音了——学习并掌握它。

《游戏音效编程》正是为此而编写的一本书。学完本书,将了解和掌握作为一名音频编程专家所应该具备的相关知识。本书并非一本高技术性书籍,它与数字信号处理、傅立叶变换、脉冲反应等知识无关。但对于有兴趣设计音效的人来说,本书是一本具有很好参考价值的基础性入门书籍。

书中将学到怎样使用 DirectAudio 的特性,比如数字文件播放、DirectMusic 及音频脚本。此外,也将学到如何播放 MP3 文件,以及下一代格式音乐 Ogg Vorbis。

本书也将涉及怎样在应用中播放 CD、混音、时间处理等细节。另外,还涉及 3D 声音建模、OpenAL 使用、音频可视化及如何使用工具创建声音和音乐。

本书最后讨论了内容创建、授权，以及创建完整声效和音乐的策略。作为希望了解更多音频编程知识的游戏音频开发者、或者希望自己书写 MP3 播放程序的业余爱好者，本书都可能成为你的一本重要参考书。我确信你无法找到另一本与此主题相关的、浅显易懂而富含实例的相同书籍。作者 Mason McCuskey 花费了大量的时间和精力，使得本书更有趣、更有价值、更引人入胜，更不用提他所创建的有趣音效了。请记住，不要听我的，听你自己的。

诚挚的



André LaMothe

Premier 出版社游戏开发丛书编辑

内容简介

开发游戏时，人们常常忽视游戏的音效。开发者往往把主要精力花费在游戏的图像和动画等方面，而忽视了背景音乐和声音效果。当他们意识到这一点时，通常为时已晚。这种做法显然是不正确的，因为好的游戏音效和音乐可以使玩家融入游戏世界，并与其情绪产生共鸣。音效的作用还不仅限于此。如果没有高超的游戏音效的映衬，再好的图像技巧也无法使游戏表现摆脱平庸，对玩家没有足够的吸引力。

以前，关于游戏音效的书很少，因为那时游戏音效设计都很简单。采用20世纪90年代的声卡，只能让计算机说一段话或播放一段简单的单声道音乐，不言而喻，其实现方法也就比较简单。今天，3D音效、环境音效、MP3播放以及CD音效等技术为游戏音效的开发提供了方便的音效技术支持(即使是设计简单游戏)。

当游戏程序员为游戏设计音效时，会面临许多技术问题。本书的编写目的就是针对这些问题，试图教会读者设计游戏音效的一些基本技能。

全书分为两个部分，第一部分讲述在诸如播放音效、装载并分析WAV文件、以及播放不同格式音乐(如MIDI、MP3、CD)等简单设计任务中，如何克服实施中的技术困难。第二部分讲述一些高级的音效技术。在这一部分中，读者将学习如何合成动态音乐并将其应用于游戏中；也将学习关于3D音乐编程、环境音效、音效脚本设计、音乐特效处理及音效可视化方面的基础知识。此外，专门有一章讲述如何使用DirectPlay语音软件在游戏中实现实时声音对话。

本书的绝大部分内容都基于DirectX及DirectX Audio技术，少数章节基于OpenAL Audio或普通Win32音频API。所有代码都采用C++编写，并基于标准模板库(STL)的低层数据结构(如向量、映射等)。

为了学习和理解本书的内容，必须了解C++，虽然并不需要精通C++；其次，还需要知道Windows程序是如何工作的；另外必须知道事件驱动的程序设计以及Win32 API的应用，包括如何书写按钮、编辑框、列表框等的GUI(图形用户接口)代码；也必须掌握DirextX的基本知识，熟悉如何获取和释放不同DirextX组件的接口；理解DirextX的功能和用途以及当你需要时到哪里获得帮助。

如果具备了以上所有知识，请从第一章开始逐章依次学习本书的内容。

致谢

许多人都为本书的编辑和出版做出了贡献。首先，我想感谢丛书编辑 André LaMothe，以及责任编辑 Jenny Davidson 和技术编辑 Heather Holland，他们帮助我将本书写得更加一致和连贯。同时，我也要感谢 Jeremy Martin 为光盘中的示例音乐所做的贡献。当然，我还要感谢我的家人和朋友们——谢谢你们！

作者

Mason McCuskey 是一位有 8 年编程经历的软件工程师，他于 1990 年成功创立了自己的游戏公司 Spin Studios。2000 年，在游戏开发者协会举办的第二届游戏程序竞赛（Game Developer's Conference 2nd Annual Independent Games Festival）中，Spin Studios 公司入围前 9 名进入决赛。Mason 也是 Premier 出版的《DirectX 特效游戏编程》（《Special Effects Game Programming with DirectX》）一书的作者，他还为 gamedev.net 和其他 Web 网站写了许多文章。

目 录

第 1 章　音频基础知识 …… 1
1.1　音效原理及术语 …… 2
1.2　游戏音效发展简史 …… 7
1.3　购买或设计音频引擎 …… 13
1.4　本章小结 …… 15

第 1 部分　音频引擎基础

第 2 章　DirectX Audio 介绍 …… 19
2.1　DirectX Audio 基础 …… 20
2.2　如何配置 DirectX Audio …… 23
2.3　音阶生成器的示例程序 …… 26
2.4　本章小结 …… 37

第 3 章　播放 WAV 文件 …… 39
3.1　用 DirectMusic 处理声效 …… 40
3.2　音频引擎中错误处理 …… 42
3.3　设计音频引擎 …… 46
3.4　播放音频的其他工具 …… 56
3.5　本章小结 …… 57

第 4 章　加载 WAV 文件 …… 59
4.1　剖析 WAV 格式 …… 60

4.2 加载 WAV 的其他方法 …… 67
4.3 本章小结 …… 72

第 5 章 声效控制技巧 …… 73
5.1 连续播放音频 …… 74
5.2 音频实例程序 …… 75
5.3 循环播放音频 …… 78
5.4 用 DirectX Audio 控制音量 …… 79
5.5 用混音器 API 控制音量 …… 84
5.6 本章小结 …… 88

第 6 章 播放 MIDI …… 89
6.1 MIDI 工作原理 …… 90
6.2 类的变更 …… 91
6.3 加载和播放 MIDI …… 93
6.4 改变音乐节拍 …… 95
6.5 检测音乐节拍 …… 99
6.6 用 Windows MCI 播放 MIDI …… 104
6.7 本章小结 …… 104

第 7 章 播放 MP3 和 WMA …… 105
7.1 播放 MP3 …… 106
7.2 播放 WMA …… 115
7.3 本章小结 …… 116

第 8 章 播放 Ogg Vorbis …… 117
8.1 Ogg Vorbis 类库结构 …… 119
8.2 安装 Vorbis …… 120
8.3 使用 Vorbis …… 121
8.4 本章小结 …… 125

第 9 章 播放 CD 音频 …… 127
9.1 用 MCI 播放 CD 音频 …… 129

9.2 CD 播放器示例程序 …… 142
9.3 本章小结 …… 143

第 10 章 播放轨迹音乐 …… 145
10.1 轨迹音乐基础 …… 147
10.2 用 MikMod 播放轨迹音乐 …… 148
10.3 本章小结 …… 150

第 2 部分 高级音频引擎功能

第 11 章 动态音乐 …… 153
11.1 动态音乐基础 …… 154
11.2 轨迹音乐与 DirectMusic 术语对照 …… 162
11.3 动态音乐编程 …… 166
11.4 本章小结 …… 172

第 12 章 脚本和特效 …… 173
12.1 音频脚本 …… 174
12.2 音频特效 …… 191
12.3 本章小结 …… 202

第 13 章 DirectX Audio 中的 3D 音效 …… 203
13.1 3D 音效的概念 …… 204
13.2 3D 音效编程 …… 210
13.3 外层空间的示例程序 …… 214
13.4 本章小结 …… 218

第 14 章 OpenAL 中的 3D 音效 …… 219
14.1 OpenAL 与 DirectX Audio 中 3D 音效的区别 …… 220
14.2 将 OpenAL 集成到音频引擎 …… 224
14.3 用 OpenAL 的外层空间示例程序 …… 230
14.4 本章小结 …… 231

第 15 章 3D 音效中的高级技术 …… 233
15.1 环境建模 …… 234

15.2 环境音效扩展技术 EAX …… 243
15.3 杜比数字编码技术 …… 244
15.4 本章小结 …… 244

第 16 章 DirectPlay 语音 …… 245
16.1 DirectPlay 语音基础 …… 246
16.2 DirectPlay 速成 …… 249
16.3 使用向导 CNetConnectWizard …… 254
16.4 DirectPlay 语音编程 …… 257
16.5 作为 3D 音效缓冲区的语音 …… 261
16.6 本章小结 …… 261

第 17 章 音频可视化 …… 263
17.1 离散傅立叶变换 …… 264
17.2 示例程序 …… 269
17.3 图形渲染 …… 273
17.4 快速傅立叶变换 …… 274
17.5 本章小结 …… 274
结束语 …… 275

附录 A 创建动态音乐 …… 277
A.1 安装 DirectMusic Producer …… 278
A.2 创建静态音乐 …… 280
A.3 动态音乐 …… 294
A.4 小结 …… 302

附录 B 创建游戏音效的体会和技巧 …… 305
B.1 音乐 …… 306
B.2 声效 …… 307
B.3 语音 …… 309
B.4 小结 …… 310

第1章

音频基础知识

这一章是游戏音效的入门知识。为了确保后续章节的学习效果,本章暂时没有具体示例程序,而只有大量的概念和术语。

在这一章中,将学到数字音频的核心知识——计算机如何听和唱;将了解关于视频游戏音频的发展简史;还将知道使用授权软件或免费下载软件来设计游戏引擎的方法。

为了避免读者花了几周时间学习本书之后抱怨:“原来我可以下载一个引擎!”,在本章中将先给出一张游戏引擎表。

1.1 音效原理及术语

所有的游戏声音都是以一个单字节流结束。因此,本书所讨论的内容以及游戏音效本身,都是基于构造大量的一维字节数组的。听起来似乎很简单,事实也确实如此。然而,这个简单的原理有时却被大量的术语搞得含糊不清。考虑到这点,这里先介绍一些基本的音效术语。

1.1.1 采样频率及采样质量

采样频率的概念贯穿音频程序设计的始终。要明白这个概念,需要首先了解数字音频是如何录制并最终存储到计算机硬盘上的(参见图 1.1)。

假设你正在制作一个动作游戏并需要一个令人毛骨悚然的尖叫声音。你找到朋友或同事,说服他坐到麦克风前,然后点击音频软件的“录音”按钮并暗示他开始尖叫。

当你点击“录音”按钮后,音频软件启动声音硬件设备开始捕获声音。通常,硬件系统捕获声音的能力取决于声卡上一块称为模拟-数字转换器的 ADC 芯片的性能。ADC 芯片时刻观察着从麦克风传过来的电流以确定声音有多大,然后,它把这个数字传给声卡设备驱动程序,并最终透过 Windows 到达音频程序,最后音频程序再将这个数字写入内存或硬盘。这个数据片断称作一个采样。(通常整段录音或采样集合也被称作一个采样,所以,必须根据上下文确定其实际所指)。

术语采样频率是指每秒钟采样的次数。采样频率以赫兹(Hz)为单位,这个单位以秒为计量基准。即 8 000 Hz 表示每秒发生 8 000 次。与字节和千字节的关系类似,频率单位也有赫兹和千赫兹(kHz),如 22 kHz 表示每秒发生 22 000 次,8 000 Hz 等于 8 kHz。

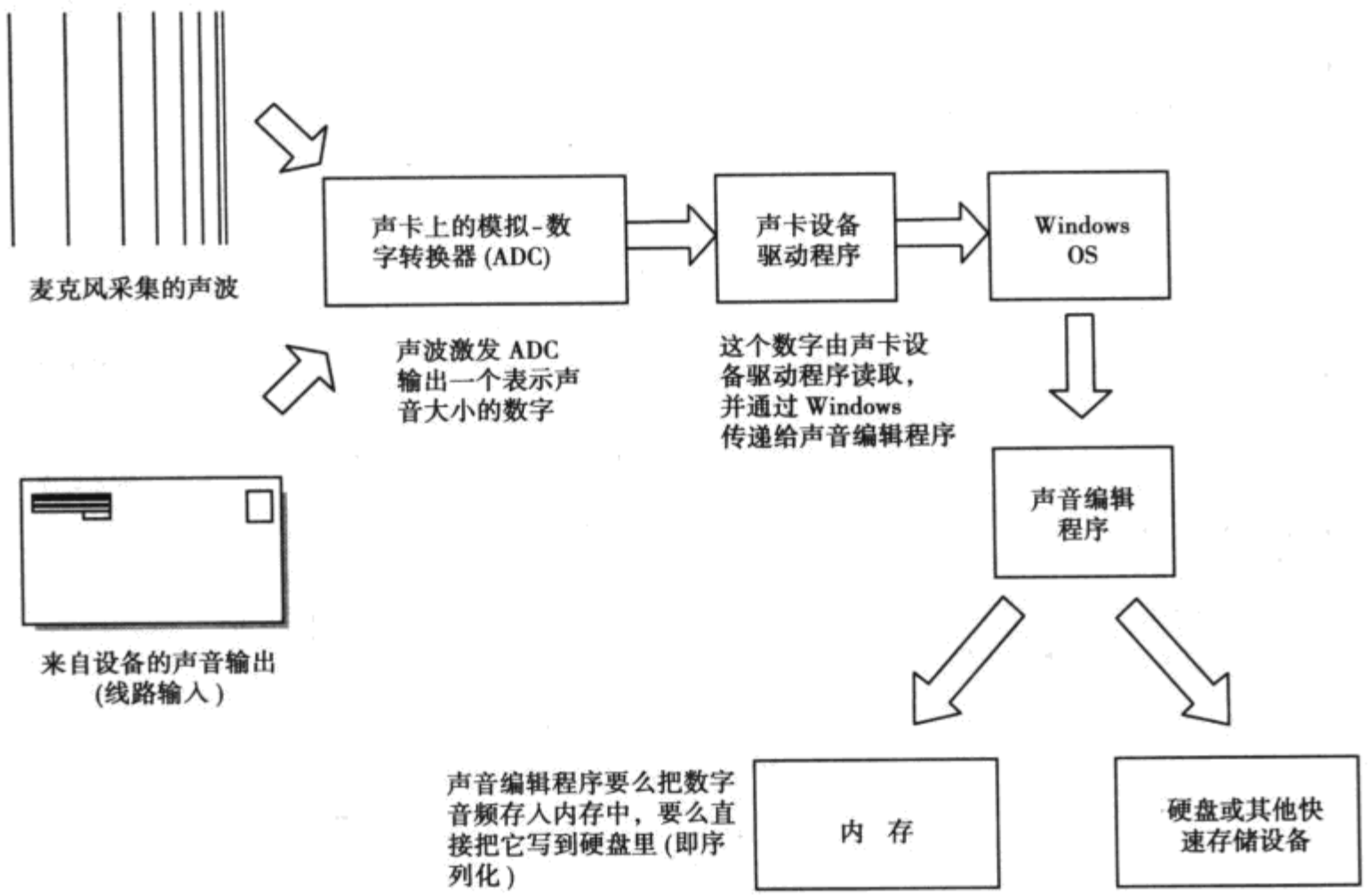


图 1.1 计算机如何录制声音

从某种意义上理解，采样频率对于音频编程如同分辨率对于图像编程。我们知道：图像的像素越多，所表示的视频图像质量就越高；同样，声音的采样频率越高，所表现的声音品质也越好。一般声卡都支持以 11 025 Hz、22 050 Hz 或 44 100 Hz 等标准采样频率来录制声音。CD 音频的采样频率是 44.1 kHz，如果要录制一个 CD 音乐节目，则计算机将以每秒 44 100 次的速度来测量空气中的声音，这将涉及庞大的数据量。

音频剪辑

播放音频剪辑 1.1 和 1.2，区分 44.1 kHz 与 8 000 Hz 采样频率的差别。

当播放音乐时，计算机读取这些数字并回送给声卡（见图 1.2）。

现在介绍影响音效的另一个术语——采样质量。采样质量（也称每个样本的比特数——bits per sample，简称比特 bits）描绘每个声音采样的精确程度。采样质量通常为 8 比特或 16 比特，8 比特可以表达 256 种不同程度的音高，16 比特却可以表达 65 536 种音高。显然，16 比特比 8 比特声音质量高，它能够更加细致地表现声音。极端情况下，如果仅有 1 比

特的采样质量,则只能表达“喧闹”和“安静”两种状态,这无法较好地表达一个声音样本。如果说采样频率类似于图像分辨率,则声音样本的比特数就类似于表示图像像素颜色的位数——比特数越大,所表示的声音就越准确。

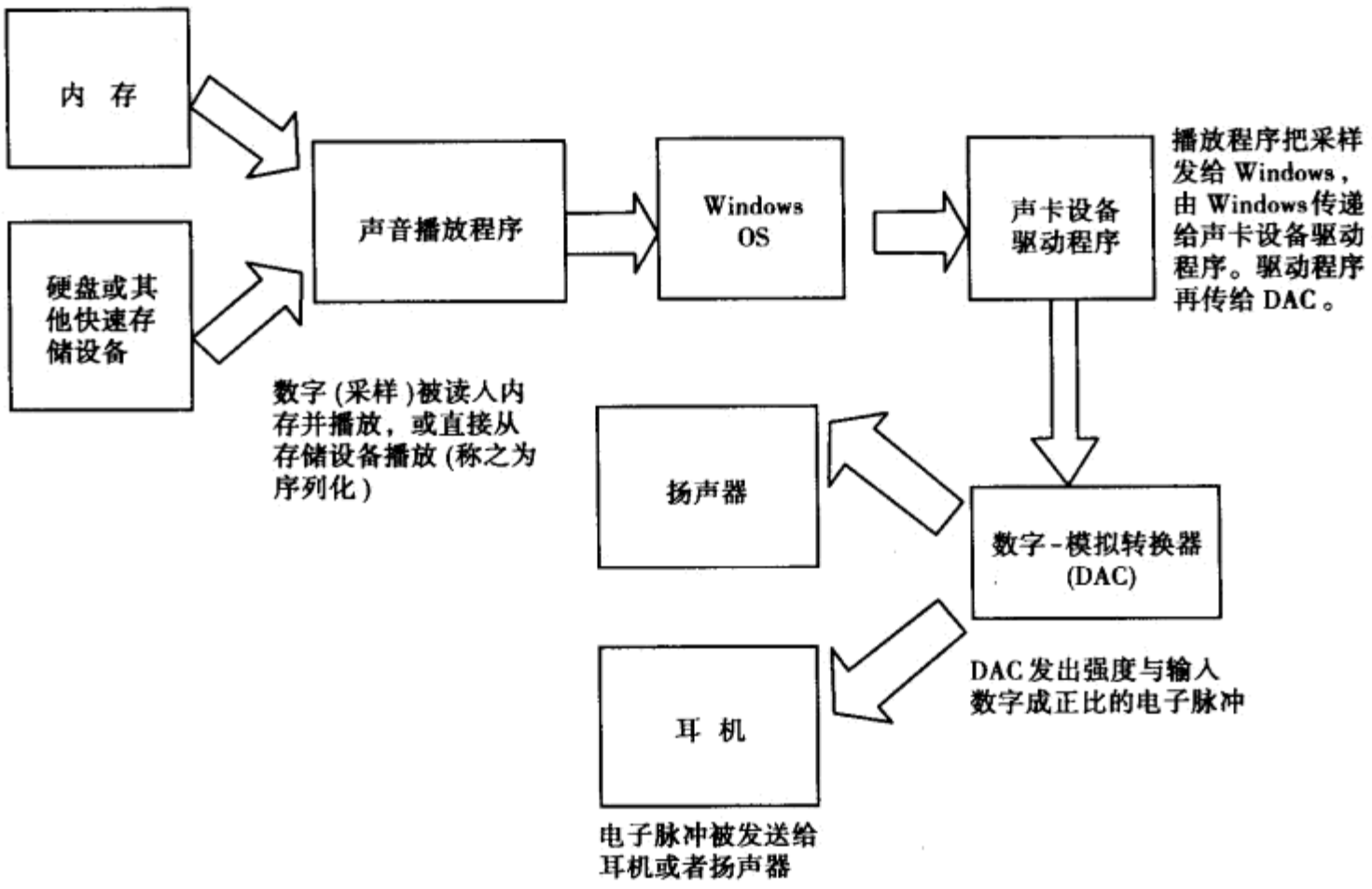


图 1.2　你的计算机如何播放声音

为什么是 44 100 Hz?

为什么 CD 的采样频率选择 44 100 Hz? 为了回答这个问题,我们需回顾音频技术的发展历史。在 CD 出现前,数字音频被存储在视频录像带上——这在当时的一种最好选择。录像带被设计成用于存储和播放每秒 60 帧的视频数据,由于每个视频帧有 245 条扫描线,而每条扫描线上都有红、绿、蓝 3 个样本。所以,总共有 245 × 3 × 60 = 44 100 个样本。

1.1.2 立体声音

到目前为止,我们只谈到一个声道的声音,也称单声道声音,简称单声道。

为了获取立体声,需要在不同的位置放置两个麦克风(一个在声源的左边,另一个在右边)。但在实际中,很少采用这种方式录制立体声音,而是使用混音器或混音软件来控制录音。混音器接受单声道的声音样本后,控制从指定声道的扬音器输出声音样本。

音频剪辑

播放音频剪辑 1.3 和 1.4,区分 8 位采样与 16 位采样的细微差别。

一个关于 CD 的故事

我的岳父(一个音乐家)曾告诉我说,CD 之所以选择存储 74 分钟音乐,是因为人们希望能不间断地欣赏贝多芬的《第九交响曲》。工程师通过计算发现《第九交响曲》的长度为 74 分钟,便定义了可以存储如此容量数据的光盘的物理指标。一个数百年前逝世的古典音乐家却能够影响今天无处不在的音频技术,真是非常有趣。

不论通过什么方式,最终是要获得有两个声道的立体声音样本。CD 音质即是指采样频率为 44 100 Hz 的 16 位立体声。对于 CD 音质的声音,每个声道每秒钟有 44 100 个样本,每个样本用 16 位(2 个字节)来记录,立体声有两个声道,所以,每秒钟的 CD 声音有 44 100 样本/声道 ×2 字节/样本 ×2 声道 =176 400 字节(约 176 kB)。这也说明了“为什么一张存储容量为 650 MB 的 CD,最多存放 74 分钟 CD 音乐”的原因。

1.1.3 声音格式、压缩及编码

当然,如果我们仅能用每秒 176 kB 的庞大数据量来存储 CD 音乐的话,数字世界无疑将会令人失望。一段 3 分钟长的歌曲将有 31 680 kB(约 30 MB)的信息量! 如果真是这样,唱片商会十分高兴,因为他们不用担心人们在线交换音乐。即使是在高速宽带连接下,为了 3 分钟音乐而传输 30 MB 数据量实在令人难以接受。

聪明的人类想出了多种音频压缩算法来处理这个问题。这些算法用更紧凑的方式表达同样的数据,从而可以用更少的字节存储相同的信息。举个简单的例子,假设有一个单声道 WAV 文件,里面存有两秒钟的寂静之声。以前,如果使用 CD 音质,这些声音将会用 88 200 字节的相同的“0”值来表示。然而,可用如下方式压缩它们:“后面 88 200 个字节都是 0”,而不需要重复地把每个“0”都写一遍。这是一种最常用的行程编码(RLE)算法的工作原理。

RLE 是一种非常简单的压缩算法,详见图 1.3。另一种更好的流行音频压缩算法是 MPEG Layer 3,用 MPEG Layer 3 压缩的声音文件以 MP3 为扩展名,其他压缩算法几乎都已

成为历史。MPEG 是移动图象专家组(Moving Picture Experts Group)的缩写,要了解 MP3 格式的历史及其创始人 Fraunhofer IIS-A,可访问 http://www.iis.fhg.de/amm/techinf/layer3/。

MP3 是一种有损压缩算法,即将一个 WAV 文件压缩成 MP3 时有信息丢失。MP3 算法可保证丢失的那些信息是不重要的信息。例如,铙钹声(cymbal)、撞击声(crash)、嘶嘶声(hiss)等中的高频信息可以丢弃。这些信息的丢失通常不会被普通听众所觉察,但如果有好的扬声器和敏锐的耳朵,也能觉察到存在微小的差异。有空可以试试——到一个安静的地方,先用 CD 听你最喜欢的歌曲,然后再用 MP3 听。如果音响很好,且你有很好的乐感,就会听出这种差异。

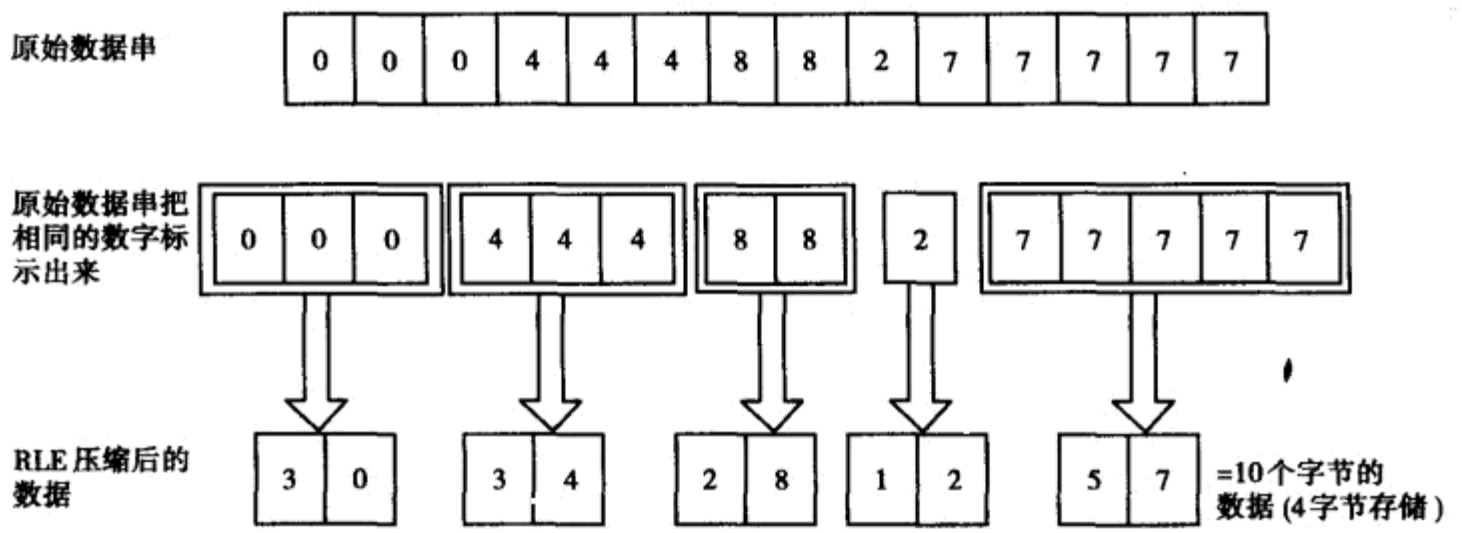


图 1.3 最简单形式的 RLE 压缩算法

目前还公布了一些其他压缩格式,如高级微分脉冲编码调制算法 ADPCM(Advanced Differential Pulse Code Modulation)、音频压缩算法 Ogg Vorbis 等。

实现这些算法的代码称为多媒体数字信号编解码器(CODEC),它是压缩器/解压缩器(compressor/decompressor)的缩写。WAV 文件记录了使用哪种 CODEC 进行压缩编码的信息;Windows 支持多数常用的 CODEC 格式,并根据 WAV 文件中的标记选择相应的 CODEC 格式进行解码。如果试图播放一个 WAV 文件,而它所标记的 CODEC 并没有在系统上安装,那么将无法播放。但这种情况并不常见,因为 99% 以上的 WAV 文件都采用 PCM 或 ADPCM 格式。

1.1.4 混音

游戏的音效并不仅限于播放 WAV 文件,还可根据需要同时播放多种音效,这就是混音

需要解决的问题。

同时播放两种声音的最简单方法是把二者的样本叠加在一起,见图 1.4。所以,如果有一个尖叫的声音样本和一个咆哮的声音样本,把它们叠加在一起播放,将同时听到尖叫声和咆哮声的混合音。尽管现实中可能不是这样,但你所听到的确实如此。

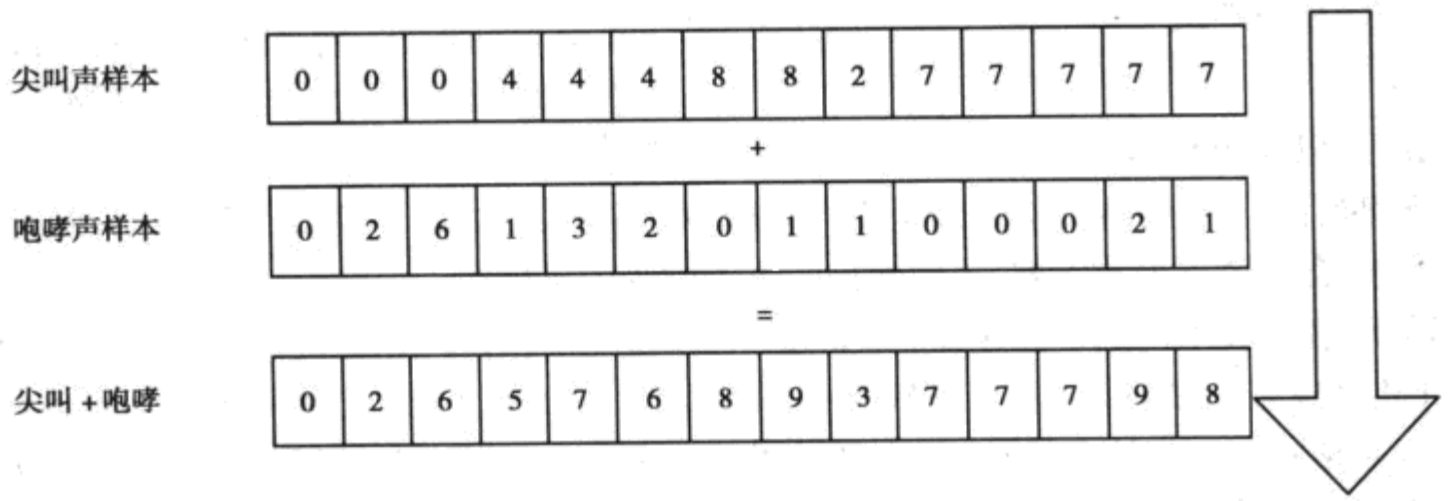


图 1.4 简单加法混音

当然,仅仅把样本累加在一起并不能获得好的结果。那么,求它们的平均值如何呢?这样做存在的问题是把每个单一的声音变弱了。如果正在同时播放 3 个音效,那么每个音效将是其单独播放时音量的 1/3。

对此有多种解决方法,如查表求对数,或使用更复杂的数学方法。事实上,如果不是专门从事音效处理的专家,就根本不需要担心混音处理的问题,因为混音处理将由 Windows 操作系统自动完成。你只需要说:“DirectX Audio(或 Windows),把这个声音的样本同那个声音的样本混合起来。”然后,DirectX Audio(或 Windows)软件将帮你完成具体的混音处理。

1.2 游戏音效发展简史

近年来,游戏音效发展很快。过去,由于技术的限制,要将声音样本用于游戏音效中是十分困难的。在 20 世纪 80 年代,一台最高档的游戏机也只有 8 kB 到 128 kB 的内存,而即使质量很差的普通游戏音效也需要有 30 ~ 50 kB。

音频剪辑

播放音频剪辑 1.5,听听早期的调频(FM)合成效果。

音频剪辑

播放音频剪辑 1.6,听听早期轨迹音乐 MOD,然后再播放音频剪辑 1.7,听一下稍迟一点的音乐。

早期的 PC 音效使用频率调制(FM)合成技术,记得我购买的第一块声霸卡就声称它支持 FM 合成。FM 合成技术使用某种算法来重建声波,它并不存储实际的样本数据,而是用数学近似的方法重建不同形状、频率和音量的声波。如果将某种频率的声波与另一种频率的声波合成,可以获得近似的喇叭声音。但是,对于诸如喇叭、小军鼓等乐器,这种合成效果并不十分理想;对于长笛或其他声波呈现纯正弦波的乐器,合成的声音效果还不错;而对于其他发声体,合成效果都非常差,采用廉价的电子键盘效果更糟。

随后,音频技术得到了迅速发展,Amiga 把 MOD 格式的音乐推向普及。MOD 音乐明显优于 FM 合成音乐,MOD 不同于 FM 合成采用数学近似的方法,它使用真实的声音样本。MOD 音乐格式最初支持 4 轨迹,每个轨迹都可以在任何时间、以任何音调播放一个样本。此时,计算机已有足够的内存来存储乐器的声音样本,并有足够的能力实时处理这些乐器的音谱。大多数样本都采用 11 kHz 或 22 kHz 的采样频率,像鼓声等乐器,因为样本是从真实鼓声录制而成,其声音质量远远超过了 FM 合成的质量。

这种技术就是波表合成(wave table synthesis)技术的最初形式。波表合成技术基于采样波形表获得所创建乐声的名字,图 1.5 给出了 FM 合成与波表合成的过程比较。

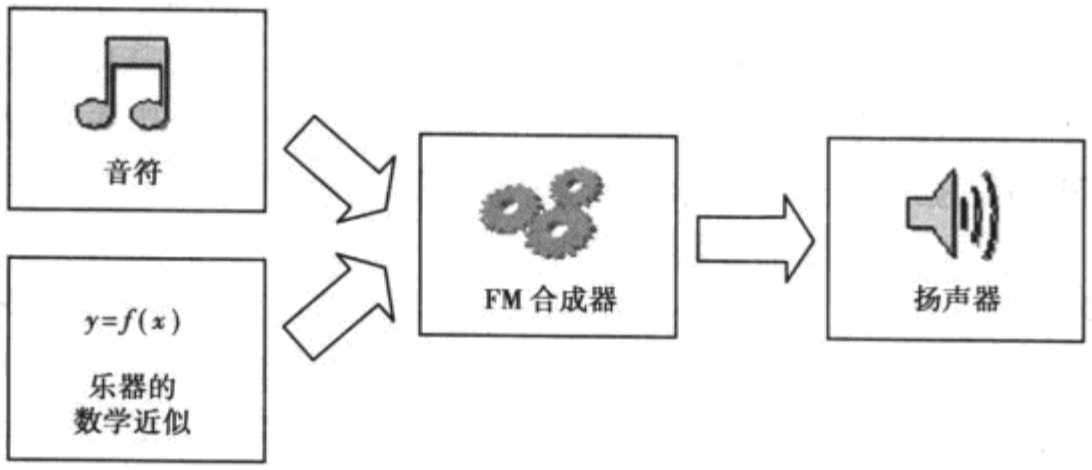
提示

MOD 音乐的继承格式今天仍在使用中。通常,在线游戏使用某种形式的 tracked music,因为这种格式远比样本歌曲来得小。一个标准的 tracked music 片断只有 200 ~ 300 kB,而一个相似的 MP3 的大小却是用 MB 来计算的。

轨迹音乐的声音非常独特。游戏的开发者常用它来播放技术性歌曲。我在本书所附光盘中附带了小量的轨迹音乐,包括我近期自己创作的一些歌曲。轨迹音乐现在依然常见,所以我也附加了几个轨迹音乐网站的链接。

关于哪一种轨迹音乐(tracked music)格式最先被提出,还存在着争论。术语“轨迹音乐”提出后,先后出现了若干种轨迹音乐的版本:StarTrekker, ProTracker, NoiseTracker, Oktalyzer 等。一个称为 Future Crew 的团队发明了 STM(Scream Tracker)和后来的 S3M(Scream Tracker3)音乐格式,并将其推广。这种格式允许更多的声道、更好的声效和更高的采样频率。Impulse tracker 进一步推进了轨迹音乐,它为视频游戏带来了更高级的音频采样能力。20 世纪 90 年代初期到中期,是轨迹音乐最为流行的时期。

FM 合成：



波表合成：

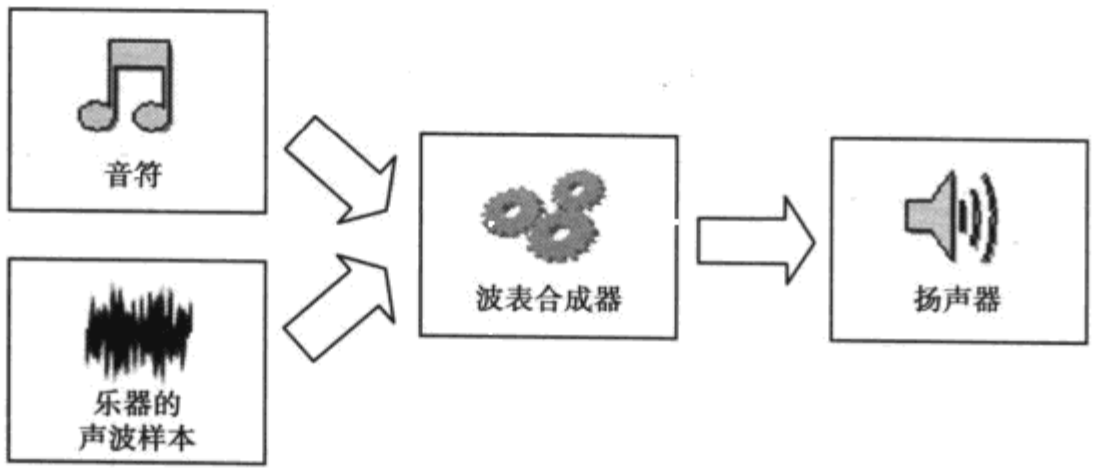


图 1.5 FM 合成与波表合成比较

1.2.1 MIDI 格式

在 FM 合成和轨迹音乐发展的同时，MIDI 技术也得到迅速发展。MIDI 是音乐乐器数字接口(Musical Instrument Digital Interface)的简写。1983 年 MIDI 1.0 问世并迅速被专业数字音乐家采用。随 MIDI 合成器价格的下降，业余爱好者和普通公民也开始使用 MIDI。

MIDI 与轨迹音乐非常相近。轨迹音乐是一个封闭系统，它由 CPU 完成所有工作：存储乐器样本，存取音符，运用声效，通过扬声器播放音乐。相反，MIDI 的原理是协调多种设备一起工作以创建美妙的乐声。一套标准的专业 MIDI 装备由具有不同用途的多台设备组成。比如，一套 MIDI 装置可能包括一个鼓机、一个合成器、多种乐器库，一台计算机和几个声效处理器，这些设备相互协调工作，以创建 MIDI 音乐。

实际上，MIDI 可以理解为是一种特殊设备的网络通信协议，它规定音符如何存储，以及

如何通过电缆传递给其他不同设备。通常,MIDI 接口规定了 128 种不同的乐声,从长笛、小提琴到枪击、直升机等各种特效声音。这 128 种不同乐器声音被称为通用 MIDI 一级声音集,简称 GM 声音集。当 MIDI 合成器接到信息"按这个音量、用 47 号乐器播放 B 调第 5 音阶"时,它知道编号 47 是哪一种乐器,并能重建相应的音色。GM 声音集如同 MIDI 的 ASCII 字符集。值得注意的是,尽管 MIDI 标准为不同乐器指定了编号,却并未指明各种乐器如何发声。假设 MIDI 数据是一个钢琴家的乐谱,那么歌曲数据(即这些乐谱)就如同音阶开关命令的序列,但歌曲演奏效果的好坏却直接取决于演奏者的水平和钢琴的质量。乐器的声音由 MIDI 硬件制造商决定,多数专业硬件制造商都是对真实的乐器进行采样,并将其存入只读存储器(ROM)中。但对于普通消费者而言,由于把真实样本存储在 ROM 中将提高声卡成本,所以,多数面向普通消费者的声卡都使用 FM 合成器来重建 GM 声音集。与直接存储乐器样本的方法相比,通过数学方法重建接近乐器声音的声质显得更为简便。

表 1.1 给出了 GM 声音集中编号和乐声的对应关系。

表 1.1 MIDI 声音集分类

编号范围	乐声类别
1 ~ 8	钢琴
9 ~ 16	变音打击乐器
17 ~ 24	管风琴
25 ~ 32	吉他
33 ~ 40	低音乐器
41 ~ 48	弦乐器
49 ~ 56	合奏曲
57 ~ 64	铜管乐器
65 ~ 72	簧乐器
73 ~ 80	管乐器
81 ~ 88	合成主奏类 Synth Leads
89 ~ 96	合成垫奏类 Synth Pads
97 ~ 104	合成声效
105 ~ 112	民族乐器
113 ~ 120	打击乐器
121 ~ 128	声效及特效

1.2.2 DLS1 标准

20 世纪 90 年代，上述多种不同的游戏音效方法都在发挥作用，其中，MIDI 和轨迹音乐最为常见，它们各有优劣。

MIDI 以文件短小见长，一般小于 100 kB，因为在 MIDI 格式下，歌曲的乐器采样不会随同歌曲本身一起存储。

不幸的是，这同样意味着 MIDI 歌曲的播放质量受乐器质量的巨大影响。专业人士具有 128 个 GM 声音集乐器的高质量样本，但其他人（特别是游戏玩家）却只有这些乐器的数学近似，通常是用 FM 合成的方法。这意味着，一个 MIDI 文件在专业作曲人士的设备上听起来很棒，但在一个标准消费者（玩家）的机器上听起来却可能很糟糕。

另外一个选择是轨迹音乐。轨迹音乐的乐器样本同歌曲本身一起存储。这意味着在不同硬件上播放不会影响最终歌曲的质量。同时，轨迹音乐使用真实声音样本而不是 FM 合成，质量通常会好一些。而且音乐家们可以创建自己的乐器样本而不仅限于 128 个 MIDI 声音集乐器样本。轨迹音乐通过自定义乐器样本的方法来解决歌词演唱问题。

轨迹音乐的缺点是尺寸很大（100 ~ 200 kB 几乎占据 3.5 寸盘 1/4 的容量，而这仅仅才一首歌曲的容量）。同时，音轨音乐也占用更多的 CPU 资源，因为 CPU 需要计算样本音高并混音。

随着技术的发展，硬件速度越来越快，存储容量越来越大，这为我们把轨迹音乐和 MIDI 融合成为一种更高级的格式奠定了技术基础。1997 年，MMA（MIDI 制造商协会）推出了第一级可下载声音标准（DLS1，Downloadable Sounds Level 1）。DLS1 赋予了消费者真正的波表合成（即用一个样本波表来表示音符，并合成乐器声）。

事实上，DLS1 允许单个乐器具有多个样本，从而使 MIDI 又进一步。这些样本按“八度”划分，你可以有一个钢琴的低音阶样本，同时也可以有另一个高八度的高音阶样本。事实上，每一种乐器，都可以拥有多达 16 个八度音域。但在轨迹音乐中要做到这一点则需要一点小技巧。

DLS1 大大推进了 MIDI 作为游戏音乐的普及。DLS 的标准包括了乐器如何发声的提示，这样，MIDI 文件在消费者硬件上的发声变得与专业硬件上的发声相近。游戏音乐的作曲者在谱曲时可以放心地确认自己所听到的与玩家所听到的相去不远。另外，音乐家们也不再局限于 128 个 GM 声音集乐器，他们可以自行采样并将他们所需要的自定义乐器包含在 MIDI 歌曲中，就如同轨迹音乐那样。

微软通过在 DirectX 中包含 Roland 创建的 GM 声音集乐器样本，进一步巩固了 DLS 的统治地位（Roland 是一家备受尊敬的音乐乐器、MIDI 键盘和 MIDI 软件的制造商）。现在，由于游戏开发者都使用相同的 DirectX 软件合成器，他们所听到的与玩家所听到的得以保持一致。

1.2.3 DLS2 标准

MMA 知道何时需要做出改进。1999 年,他们批准了 DLS2 标准。针对 DLS1 的缺点,DLS2 采取了几个技术措施进行改进。在这里不详细罗列,但可以肯定地说,DLS2 优于 DLS1,它有很多让人欣喜之处。

最早支持 DLS2 标准的产品是 DirectMusic8。微软 DirectMusic 中的声音合成器能支持 DLS2,且只需耗费极少的 CPU 资源,因此,可以在速度很慢的机器上运行。

1.2.4 CD 音频标准

现在来谈 CD 音频,又称红皮书音频。对于商业游戏,CD 音频是可以通过 CD 发布的一种音乐存储格式。

提示

CD 音频之所以被称为红皮书音频是因为其原始标准来源于 Sony 的一本红色封面的标准书。红皮书音频标准就是 CD 音频标准。所有的 CD 音频都是按照 Sony 的红皮书标准制造的。如果你有一张游戏 CD,并发现它可以在你的车载 CD 播放器中播放,则这款游戏肯定是使用红皮书音频作为其音效格式。

红皮书音频使得通过 CD 发布的游戏可以避免 MIDI/轨迹音乐的缺陷。CD 上的歌曲无论在哪里播放效果都是完全一样的。CD-ROM 驱动器负责完成播放一条音轨的大部分工作,CPU 资源占用很小。同时,由于 CD 可以存储如此多的数据(650 MB),空间也不是问题。

CD 音频:即将失势?

1979 年 Philips 设计了第一张 CD 歌碟,1983 年 Sony 推出了第一款 CD 播放器(1 000 美元)。能在 CD 机上播放的第一张专辑是 Billy Joel 的《第 52 街》。1983 年全美共有 3 万台播放器和 8 万张 CD 歌碟。9 年后,增长到 920 万台播放器和 2.88 亿张 CD。

CD 的迅速普及以及它给 PC 机带来的似乎无限的存储容量,使其一度成为最佳视频游戏音频存储方式。但近年来,对于游戏开发者而言 CD 音频似乎不再那么有吸引力了,650 MB 的容量也不像过去那么大。当你可以用 MP3 表达质量几乎相同的音频,而容量仅为 CD 音频 1/10 时,你还会采用 CD 音频格式吗? 此外,CD 音乐不可能成为动态音乐。MIDI 或样本音乐可以成为动态音乐的组成部分,根据不同的游戏进程而表现出声效差异。

对多数游戏而言,CD 音频仍然是好的选择,但是,它在高倍压缩及动态音乐面前已经失势。CD 音频是否仍将作为游戏音效的常用格式,目前还是一个未知数。

1.2.5 音频格式的选择

前面介绍的3种技术(轨迹音乐、MIDI/DSL/DirectMusic、红皮书音频)今天都还在应用。FM合成依然与那些使用MS-DOS和Windows 3.11的保守者同行。

关于游戏音效,需做出的最重要决定之一就是选择何种音乐格式。下面是关于如何选择音乐格式的一个示例。示例中的MP3,可以替换成其他"你所喜欢的音频压缩格式"。示例中的DirectMusic,是指用DirectMusic构建的音乐。

首先决定使用静态音乐还是动态音乐。

如果希望音乐是动态的,DirectMusic可能是最佳选择,当然也可以通过改进一个轨迹音乐引擎使其支持动态音乐。但要得到效果与DirectMusic相当的音乐引擎需要做大量的工作。也可以自行设计一个系统并基于样本音乐片段构造动态音乐。比如,可以对同一首歌的不同变种采样,然后书写代码随时或者根据游戏进程选择播放其中一个样本。

如果需要静态音乐,则可以采用任何一种音频格式。如果你的游戏将会在线销售,强烈建议你采用容量较小的轨迹音乐。

红皮书音频以及压缩的样本音乐效果较好,但前提是必须有制作专业音质音乐的手段,或者已有写好的乐谱。如果试图考虑在低档硬件运行,红皮书音频将是个较优的选择,因为它几乎不占用CPU资源。此外,建议使用压缩音乐格式,如MP3等。开发者必须注意,红皮书音频在游戏中要求编写一个抽象层去模拟一个CD音轨,这会增加系统的复杂性。而且,在每次音乐发生变化时,必须重新烧录CD-R。

如果使用MP3,必须注意授权问题。通常情况下都可以自行编码以避免支付授权费用(之所以如此是因为用户的系统上有已授权的解码器),但这样做有可能会失去一些控制力(详见第6章,"MIDI/DirectMusic播放")。

对于商业游戏,通常在MP3(静态音轨)和DirectMusic(动态音质)之间做出选择。对于在线游戏,一般选择MP3或轨迹音乐。

本书附带光盘中有相关网站的链接,这些网站将会是我所讲述内容的补充。

1.3 购买或设计音频引擎

本书的主要目的是教读者设计音频引擎。后续的章节,将会讲解如何将一些基本的和较高级的特征赋予音频引擎。学到本书最后,将十分清楚音频引擎的工作原理并且能够让它在你自己的游戏中完全发挥其功能。

当然,并不是每个人都需要自行设计音频引擎。现代社会的美妙之处在于可以用金钱换取时间——购买现成的引擎。下面给出一些免费的、共享的或商业音频引擎,如果不希

望自己写引擎的话,可以使用它们。这些引擎按字母顺序排列。你也可以在本书配套的CD上找到几个引擎的演示版本。

> **提示**
>
> 请注意这些信息是在2003年收集的,到现在情况会有一些小变化。请访问供应商的网站以获取最新信息。

1.3.1 音频引擎BASS

免费软件
共享软件100美元
商业软件995~2 750美元
http://www.un4seen.com/music

BASS是Windows平台上一个非常复杂的音频引擎。它支持当前所有常见的音频文件格式(WAV、MP3、WMA以及Ogg Vorbis),仅需要DirectX 3的接口支持(可在NT上使用它),支持EAX及3D音频以及最常见的轨迹音乐格式(MOD、XM、IT、S3M)。

BASS另外一个长处在于它支持多语言。除了C/C++的API之外,它还有Visual Basic、Delphi、MASM和TMT Pascal的API支持。

BASS引擎已经得到广泛应用,特别是在自主开发者群体中受到欢迎。

1.3.2 轨迹音乐库MikMod

免费软件
http://www.mikmod.org

免费软件MikMod是一个跨平台的轨迹音乐库,最早由Jean-Paul Mikkers编写,现在由Miod Vallat维护。

MikMod支持所有的常见音频格式,可以在很多种不同操作系统下运行,包括Mac OS、BeOS、Java、Unix以及Windows。

1.3.3 音频引擎FMOD

免费软件
共享软件100美元
商业游戏应用1 250美元
http://www.fmod.org

FMOD 是一款非常复杂的音频引擎,在几乎每一种平台上都可用,包括 PS2、Xbox 以及 GameCube。它支持各种各样的 C ++ 编译器及 VB,包括诸如 EAX 支持、阻塞/回收等高级特性。

1.3.4 轨迹音乐播放器 ModPlug

免费

http://www.modplug.com/

如果希望以快速简单的方法播放轨迹音乐,ModPlug 就是一种最好的选择。ModPlug 有非常简单的接口和强大的播放引擎,它支持所有能见到的轨迹音乐格式。它有 C ++ 、Visual Basic、Delphi 的接口,并有 ActiveX 控件可供使用。

1.3.5 游戏音效库 Miles Sound System

一个产品许可 4 000 美元

一个站点许可 12 000 美元

一个跨平台站点许可 22 000 美元

http://www.radgametools.com

Miles Sound System 是一个 RAD 游戏工具,是目前为止最为复杂的游戏音效工具库。它最早是由 John Miles 在 1991 年的夏天发布的,目前已在超过 3 000 个游戏中得到应用。它可以做你想做的任何事。借用 Ferris Bueller 的话,“如果你有钱,极力推荐你买下它”。

1.4 本章小结

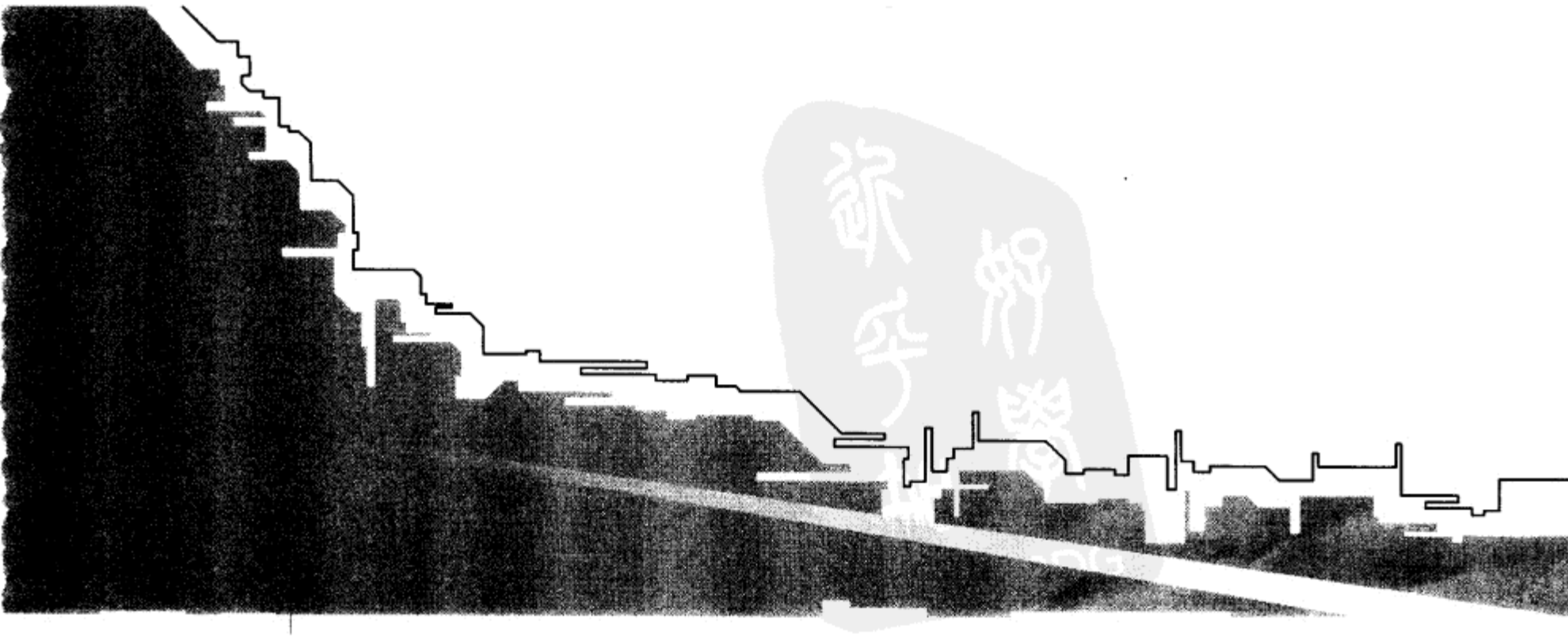
现在已经学到了书写一个音频引擎之前所必须的术语和背景知识。如果想获得更多信息,请访问本书附带 CD 里提供的链接——它们将把你带到关于此课题的所有有趣的网页去。

PDG

PDG

第 1 部分

音频引擎基础



在这一部分里，将讲授如何创建一个功能完整的音频引擎。音频引擎将用 C ++ 编写，并使用标准模板库（Standard Template Library，STL），其运行基于微软的 DirectSound 和 DirectMusic。

从这一部分的各章节里，读者将比较系统地接触音频引擎的相关基础，如：学到实现引擎的一些重要技术；学习如何把音效添加到引擎里；粗略地了解数字声音；学习底层的 DirectX 程序，学会如何把数据放入声音缓冲区以创建音效；学习如何读取和检查波形（WAV）文件，以及如何播放和循环播放它们等等。

在音效被加入到引擎里并开始生效之后，重点将转到游戏音效上，学习如何读取和播放各种音乐格式的文件。读者将学会如何播放 MP3 文件，以及如何读取和播放 MIDI 文件、Ogg Vorbis 文件和各种记录格式的音频文件（如 S3M、IT、XM 等）。

在这一部分结束时，读者将尝试设计完成一个能被游戏使用的音频引擎。如果只需要用到音频引擎的基本功能，其实现并不复杂。本部分任务仅仅是播放音效和音乐，更复杂的功能将在第 2 部分介绍。

下面进入正题。

PDG

第2章

DirectX Audio介绍

最长的旅途都始于第一个脚印，最好的音乐都始于一个音阶——440 Hz 的 A 音阶。本章通过示例程序来讲解如何产生 A 音阶。

本章介绍 DirectX Audio。如果曾用过 DirectX，那么很多内容将不会陌生——获得音频接口、设置协作模式等，本质上与它们在 Direct3D 中所对应的部分非常相似，有趣部分是后面将介绍的产生音阶的基本算法。

2.1 DirectX Audio 基础

要想理解设计给出的示例代码，必须掌握一些基本概念。因此，本节首先介绍这些相关的基本概念。

2.1.1 DirectMusic、DirectSound 和 DirectX Audio

从音频接口名称看，起初称 DirectSound，后来称 DirectMusic，再后来，微软在 DirectX 8 中提出了 DirectX Audio 的称谓。不幸的是，这 3 个术语中有两个都用词不当——可以用 DirectMusic 播放声音，也可以用 DirectSound 播放音乐，这不是用词不当吗？因此，必须清醒地认识到，这些术语并不是用来表示不同的声音类型的，决不能误认为“这个是用来处理声效的，那个是用来处理音乐的”，但可以把它们分类成低层 API 和高层 API。表 2.1 总结了这 3 个名词的不同。

微软让我们把 DirectMusic 作为音频 API，在 DirectX 9 中仍可直接访问低层 DirectSound API；但通常只使用高层 DirectMusic API，而不用关心低层 DirectSound API（除非想用低层 API 来控制 WAV 文件）。即使将来微软屏蔽了 DirectSound API，而把 DirectMusic API 作为 DirectX Audio 中惟一的接口，也不会使人感到惊讶。

表2.1 DirectX Audio术语定义

接口名称	含义
DirectSound	Windows平台提供的低层的音频API。它涉及通过声卡输入/输出原始声音数据的操作细节
DirectMusic	Windows平台提供的高层的音频API。它相对更抽象,但在不能确定是一个WAV文件、MIDI文件还是SGT(dynamic music segment)文件时,它能够加载并播放这个文件
DirectX Audio	微软对整个音频API的统称——如DirectSound和DirectMusic都包含在其中

但是,本章剩余部分将重点介绍低层接口DirectSound,不是因为它比DirectMusic简单或者它是更好的接口,而是因为DirectSound可以直接处理声卡的数据。对于刚开始进行游戏音效编程的爱好者而言,它是最简单可行的方法;对于已有游戏音效编程经历的开发者而言,它可能是最常见的方法。

2.1.2 主缓冲区和辅缓冲区

作为DirectSound的核心,DirectSound依赖于主缓冲区和辅缓冲区的概念。这里所说的缓冲区是指一个一维字节数组,这些字节将最终变成音频。

每个应用程序都有一个DirectSound主缓冲区,扬声器根据主缓冲区中的内容周期性地产生音频。

每个应用程序至少需要一个DirectSound辅缓冲区,根据需要也可以有多个辅缓冲区。辅缓冲区保存单一的声音或音频数据流。

这些缓冲区相互配合,如图2.1所示,实质上就是把各种声音存储在辅缓冲区中,然后将它们合成之后放入主缓冲区中。通常,由DirectSound负责处理合成操作,因此,操作者实际上不用自己去管理主缓冲区。当然也可以直接访问主缓冲区,但必须记住:如果想通过直接访问主缓冲区自己完成合成操作,将不能创建任何辅缓冲区。如果不是做很特别的事情,建议不要直接访问主缓冲区。

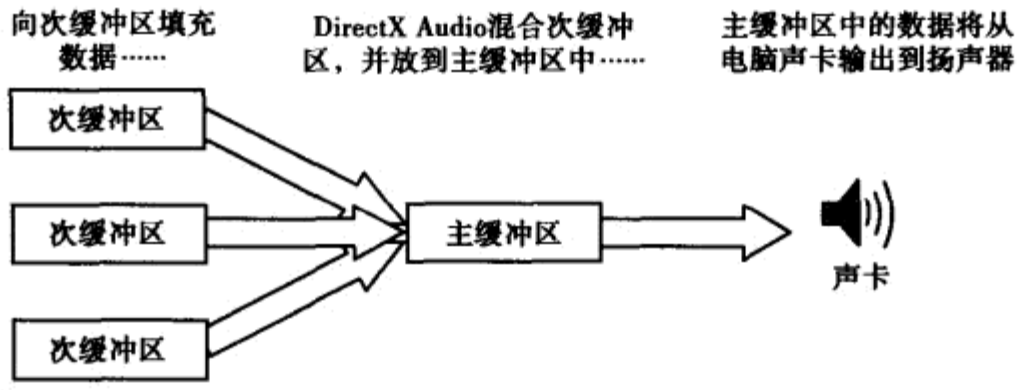


图 2.1 DirectX Audio 混合次缓冲区的数据到主缓冲区中，然后输出到扬声器

提示

现在，硬件缓冲区与软件缓冲区并没有实质性区别。但在 20 世纪 90 年代，因计算机的 ISA 总线的速度非常慢，将常用声音存储在声卡的 RAM 中效果更好。当声卡需要合成声音时，它将不必通过慢速的 ISA 总线来传输数据。

现在，PCI 总线相对来说很快，因此声卡不需要有自己的 RAM——所有的数据都存储在系统内存中，并通过快速的 PCI 总线传送数据。PCI 总线对于音频数据而言有足够的带宽（尽管对于图形数据而言还有点慢，这也是为什么显卡还带有 RAM 的原因）。

如果想了解更多的技术说明，可以通过如下网址阅读 Brain Schmidt 的文章“Sound Cards, Voice Management, and Driver Models”：

http://msdn.microsoft.com/library/default.asp?url=/library/enus/dnaudio/html/ntk_sc.asp。

缓冲区可以指定硬件区域或通过软件创建，硬件缓冲区在声卡的 RAM 中，而软件缓冲区在系统内存中。

2.1.3 流技术

前面已经介绍了声效数据将被放在辅缓冲区中。那么，当辅缓冲区较小、不能存放完整的声效数据时，该怎么办呢？例如，播放一段 15 分钟长的游戏剪辑音乐，不能一次就把整个数据都放到存储器中（即使空间够放，也不是最好的办法，因为它将占用太多的存储空间）。对于这个问题，较好的解决方法是在需要时加载相关部分的音频数据，换句话说，就是在需要时将它从硬盘中动态调入内存。

因为从硬盘或 CD-ROM 读取数据的速度远快于音频播放速度，因此，流技术已成为一种有效的常见技术。图 2.2 能帮助理解流技术的概念，随着本书学习的深入，将逐步加深对流技术的进一步了解。

当音频数据流动时，不用立刻加载所有数据，而是把辅缓冲区看成是一个在音频数据上滚动的窗口，只加载需要的部分数据。比如，在图2.2中可以看到10秒钟的音频数据（一首3分钟长的歌的开始部分）被加载到缓冲区中；然后程序指示缓冲区开始播放，音乐随即开始；当音乐在播放的时候，计算机从磁盘读取播放游标后的新数据并放在缓冲区中，覆盖掉已经播放的数据；播放游标回到缓冲区的开始处并播放新的数据。如此往复，直至整个数据播放完。

显然整个操作取决于读取数据要快于播放。令人高兴的是，如今硬盘读取数据远远快于播放它们的速度。

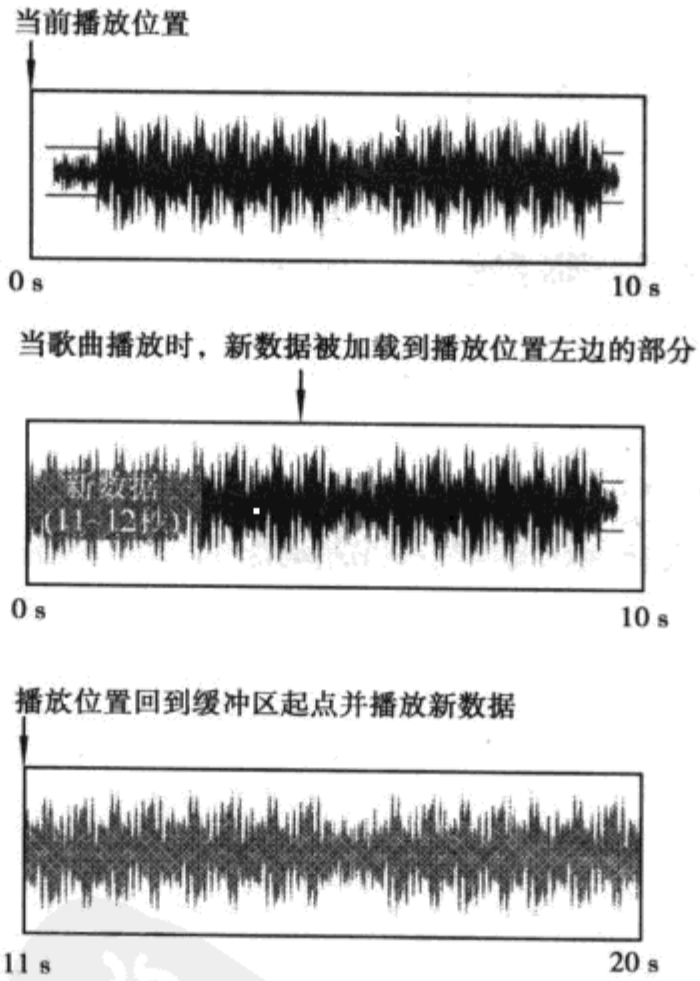


图2.2 播放一个大的声音文件流只需要加载需要的部分

2.2 如何配置 DirectX Audio

首先，应该学会如何正确地配置编译器来创建使用 DirectX Audio 的应用程序。这就需要告诉编译器在哪儿可以找到 DirectX 头文件，在哪儿可以找到 DirectX 库文件。本节教授如何完成这些事情。不论使用的是 VC ++ 6，还是使用的是其他编译器，或 Visual Studio 7，其概念都是相同的，不过只是要点击的按钮和菜单不同而已。

2.2.1 全局配置与项目配置

DevStudio 有两个不同的配置集合。一个是全局配置,它作用于所有项目,并保存在计算机的注册表中;另一个是项目配置,它仅作用于被加载的项目(保存在 DSP 项目文件中)。项目配置优先于全局配置(DevStudio 先查找所需要使用的项目配置,如果没有找到,就使用全局配置)。

2.2.2 构建示例程序

本书中的所有示例程序都假定已经在全局配置中设置了 DirectX 路径(如图 2.3)。

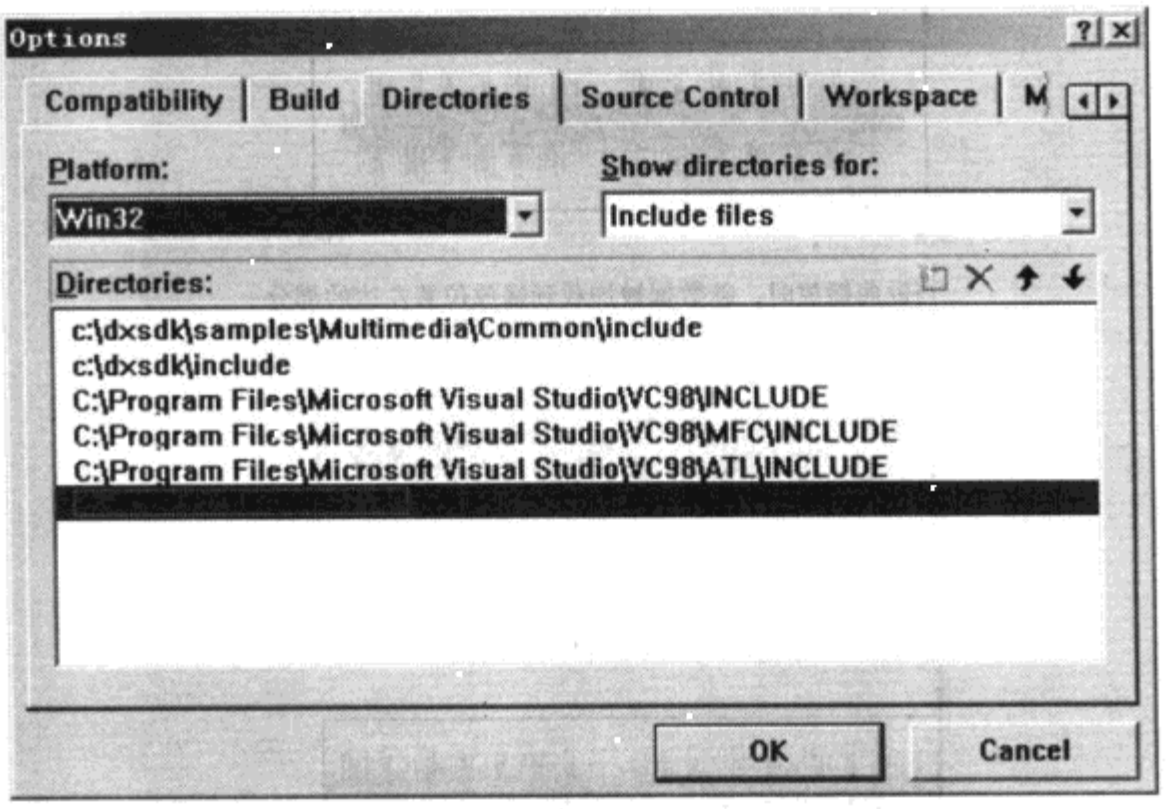


图 2.3 设置全局 include 路径

安装 DirectX 时,有一个选项可以自动添加适当的路径到 DevStudio 的配置中。如果没有选择这个选项,可以用下面的步骤来添加路径。

第 1 步:从 Tools 下拉菜单中选择 Options;然后从弹出的对话框中选择 Directories 选项(可能需要使用右边的滚动按钮来滚动选项卡)。

第 2 步:在 Show Directories For... 下拉框中选择 Include Files。

第 3 步:点击红色 X 左边的按钮来创建一个新条目,输入 DirectX include 目录的完整路径。DirectX SDK 默认安装在 C:\dxsdk 下,因此应该输入**C:\dxsdk\include**,也可以指定其他的安装路径。同样地,输入示例程序的 include 目录的路径,一般是 C:\dxsdk\samples\Multimedia\Common\include。

第4步:将这两个目录拖动到列表的顶端,使得 DevStudio 首先检查这些路径。这步很重要,因为 DevStudio 的默认安装中包含了一个老版本的 DirectXSDK,如果错误地加载了老版本,将会导致出现错误,比如"LPDIRECTSOUND 8 doesn't exist"。

第5步:从 Show Directories For 下拉框中选择 Library Files,用同样的步骤添加 DirectX library 所在的完整路径(通常是 C:\dxsdk\lib)。

第6步:点击 OK 确认,然后退出 DevStudio 并重新启动它。DevStudio 在退出时将保存全局配置,因此如果养成改变全局配置后退出的习惯,在 DevStudio 异常终止时(尽管这种情况很少发生),就不用重新改变设置。

提示

运行示例程序不用设置项目配置,但如果你想改变自己项目的项目配置,可以在 Project 菜单中选择 Settings。在 C/C++ 选项的 Preprocessor 部分里可以找到 include 文件夹,在 Link 选项的 Input 部分可以找到 library 文件夹。

现在,全部设置完毕。如果仍然遇到错误提示(如"could not open include file dsound.h: no such file or directory"),或者遇到相似的连接错误,则可能是路径设置不正确。

2.2.3 构建自己的项目

使用 DirectX 时,需要通过几个步骤来建立和运行应用程序框架。下面,以创建 Ch2p1_ToneGenerator(一个音阶生成器)的工作空间为例,讲述需要完成的基本步骤。

提示

音阶生成器 Ch2p1_ToneGenerator 工作空间在 CD-ROM 中。文中所写的这些步骤仅仅作为创建其他项目的参考。

第1步:从 File 菜单中选择 New;然后选择 Projects 选项,再选择想创建的项目类型(如,Win32 Console Application,Win32 Application,或 MFC AppWizard EXE),并输入项目名称及所在位置。

第2步:按弹出的向导框的提示进行操作。这里没有特殊的设置。对于 ToneGenerator,可以从"Hello World"开始。

第3步:按向导操作结束后,可以得到一个编译和运行都没有问题的程序。如果无法获得该程序,请告知微软他们的源代码不能编译。

第4步:从 Project 菜单中选择 Settings。在对话框左边的 Settings For 下拉框里选择 All configurations。

第5步:跳到 Precompiled Headers 部分,选择 either Not using precompiled headers or Au-

tomatic use of precompiled headers。对于小项目,从 stdafx. h 和 stdafx. cpp 文件中预编译头文件是件痛苦的事。

第 6 步:转到 Link 选项,把 dxerr8. lib、winmm. lib、dsound. lib、dxguid. lib 添加到 Object/Library modules 编辑框中。大多数的示例程序和很多个人创建的 DirectX Audio 应用程序都需要这些库文件。

第 7 步:转到 Input 部分,把 DX 库文件的相对路径加到 Additional Library Path 编辑框中。

第 8 步:点击 OK 按钮,关闭 Project Setting 对话框。

第 9 步:转到文件视图框,从项目中删除 stdafx. cpp 和 stdafx. h。在所有资源文件中(这里应该只有一个文件),删除所有包含 stdafx. h 的行。因为在第 5 步中,关掉了预编译头文件,这些文件都不需要了。

第 10 步:Stdafx. h 中包含了一些有用的行,定义了 WIN32_LEAN_AND_MEAN,并包含了 stdio. h。如果需要,可以将它们放到 C ++ 主文件中,或者是全局头文件中。

第 11 步:从硬盘上删除 stdafx. h 和 stdafx. cpp。

第 12 步:验证所有代码是可编译的。如果无法编译,很可能是在前面的步骤中出现了错误,或者发生了一些“奇怪”事情。

提示

从项目列表中选择 DirectX AppWizard,可以轻松地创建 DirectX 应用程序。AppWizard 将引导创建类似于 SDK 中 DirectGraphics 示例程序的 DirectX 应用程序。不能用向导来创建游戏,但可以用它来创建一个应用程序的框架,以便测试某些事情。

尽管已经删除了所有 stdafx. h 文件,仍有可能在 External Dependencies 文件夹中看到它。这个文件夹在每次正确编译后会更新,因此,一旦完成了项目构建,stdafx. h 就不会再显示了。

2.3 音阶生成器的示例程序

音频剪辑

播放 Audio Clip 2. 1,请听本章示例程序的输出效果。

编译器已经设置好了,现在可以开始编制 DirectX 音频程序了! 如果愿意,也可以把 Ch2p1_ToneGenerator 示例程序代码导入到自己的 IDE 中。

2.3.1 何谓音阶生成器

音阶生成器是一个简单的控制台程序,用来初始化 DirectX Audio 和演示发声的基本操作。音阶生成器教你使用 Direct XAudio 的基本操作和声音的低层核心行为。它既不太复杂,也不太实用,但它的确是一种理解"产生数字化声音的基本框架"的好方法。

音阶生成器产生一个特定频率的音阶。它计算一个正弦波的值,并把计算结果值存入辅缓冲区,然后用 DirectX Audio 合成送入主缓冲区,并输出到扬声器。它是一个控制台程序,没有 GUI 代码,也没有处理锯齿波和方波。如果不清楚正弦波、振幅、周期和频率,可以看图 2.4。下面,来看它是如何工作的。

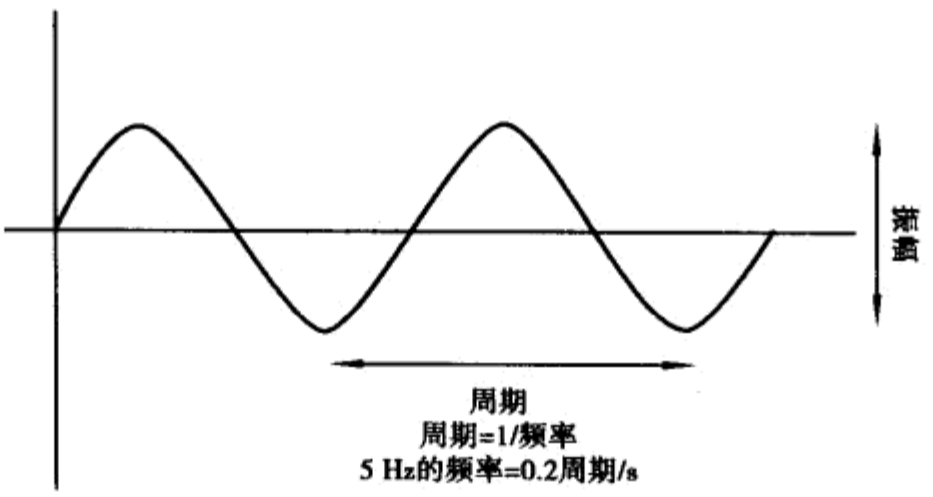


图 2.4 正弦波

2.3.2 初始化 DirectSound

初始化是第一步,也是最重要的一步。初始化包括 3 个主要步骤:获得接口、初始化协作等级和设置主缓冲区格式。

前两步比较简单,可以跳过繁琐的声卡列表,通过把 NULL 作为 DirectSoundCreate8 的第一个参数来选择一个声卡(或参考第 4 章设备列表的细节)。设置协作等级也比较简单。对于多数游戏而言,都采用 DSSCL_PRIORITY 设置主缓冲区的格式。如果要写一个音频应用程序,像 Winamp 或 ACiD,可以用 DSSCL_NORMAL,它不允许更改主缓冲区格式,但可更快地在应用程序间切换,总的说来,DSSCL_NORMAL 比 DSSCL_PRIORITY 对系统更有好处。

提示

调用 SetCooperativeLevel 时，其中一个必须的参数是应用程序主窗口句柄。因为音阶生成器是作为控制台窗口运行的，所以它使用了一些 Windows 的功能来获得 HWND。请查阅 Ch2p1_ToneGenerator. cpp 示例程序中的 GetConsoleWindowHandle 函数。这个函数查询 Windows 得到其标题(window caption)，然后找到第一个使用该标题的窗口句柄。这是一个有点绕圈子的做法，并且只能使用在只有一个应用程序实例运行的情况下。但对于示例程序而言，这种做法已经足够了。

现在就剩下设置主缓冲区格式的任务了。相对来讲，这一步有点麻烦，因为必须要获得主缓冲区的接口，其代码如下所示：

```
HRESULT hr;
LPDIRECTSOUNDBUFFER pDSBPrimary = NULL;
// 获得主缓冲区
DSBUFFERDESC dsbd;
ZeroMemory ( &dsbd, sizeof ( DSBUFFERDESC) ) ;
dsbd. dwSize            = sizeof ( DSBUFFERDESC) ;
dsbd. dwFlags           = DSBCAPS_PRIMARYBUFFER;
dsbd. dwBufferBytes     =0;
dsbd. lpwfxFormat       = NULL;

// “创建”(如果已经存在，就选中它)
if ( FAILED ( hr= g_pDS -> CreateSoundBuffer ( &dsbd, &pDSBPrimary, NULL ) ) ) {
   cerr << " CreateSoundBuffer failed: " << DXGetErrorString8 ( hr) << endl;
   throw ( err. str( ) ) ;
}
//设置格式
WAVEFORMATEX wfx;ZeroMemory( &wfx, sizeof ( WAVEFORMATEX) ) ;
wfx. wFormatTag         = WAVE_FORMAT_PCM;
wfx. nChannels          = ( WORD) 2;
wfx. nSamplesPerSec     =22050; // 22 Khz (CD quality is ~ 44 Khz, so this is "half-CD"
                          quality)
wfx. wBitsPerSample     = ( WORD) 16;
```

```
wfx.nBlockAlign        = (WORD) (wfx.wBitsPerSample / 8 * wfx.nChannels);
wfx.nAvgBytesPerSec = wfx.nSamplesPerSec  *  wfx.nBlockAlign;
if(FAILED(hr = pDSBPrimary -> SetFormat(&wfx))){
   cerr << "pSDBPrimary -> SetFormat()failed: " << DXGetErrorString8(hr) << endl;
   throw(err.str());
}
// 做完后释放资源! SAFE_RELEASE( pDSBPrimary );
```

这段代码执行了3个步骤。首先是获得主缓冲区。它使用的方法是调用CreateSoundBuffer。CreateSoundBuffer带有一种结构,并根据结构中的标志来确定:是创建一个辅缓冲区并返回其接口,还是返回主缓冲区的接口。这里,我们得到主缓冲区并存储在pDSBPrimary中。

第2步是用所希望的音频格式属性来创建WAVEFORMATEX结构。这里,属性是双声道(立体声),22 050 Hz(半CD音质),每个采样点16位编码。这就意味着,对于每一秒钟的音频,我们有22 050个采样点,并且每个采样点有16位(2字节),因此每秒钟的音频需要44 100字节的存储空间。22 050 Hz是"平均质量"的采样频率;作为比较,CD上的音乐的采样频率是44 100 Hz。表2.2列举了一些常见的采样频率。

表2.2 常见的音频采样频率

采样频率	Windows名称	位	声道
8 000 Hz	N/A	8	1(单声道)
11 025 Hz	电话质量	8	1(单声道)
22 050 Hz	广播质量	8	1(单声道)
44 010 Hz	CD质量	16	2(立体声)

一旦正确创建wfx结构后,代码调用主缓冲区接口的SetFormat,使主缓冲区转化成wfx所指定的格式。

最后,代码调用SAFE_RELEASE宏,释放主缓冲区接口。注意,这里并没有物理删除主缓冲区,它只是释放CreateSoundBuffer所创建的接口。一个DirectX对象(或任何COM对象)只有当它的所有接口都被释放后才会被删除,而DirectX Audio保持了一个主缓冲区的内部接口,所以直到代码反初始化DirectSound,主缓冲区才会被删除。

如果代码没有释放接口，主缓冲区永远也不会被删除。请牢记在做完对主缓冲区的操作以后释放主缓冲区接口。实际上，DirectX Audio 能够自动检测出是否发生这种情况，一旦发生就会自动修复。

2.3.3 创建辅缓冲区

DirectSound 被初始化后，下一步是创建一个辅缓冲区。辅缓冲区是保存实际音频数据的地方；DirectSound 利用主缓冲区将所有的辅缓冲区合成在一起并输出到声卡。令人高兴的是，由于函数调用是一样的，所以创建一个辅缓冲区的 90% 的内容实际上已经了解了。调用 IDirectSound8 的 CreateSoundBuffer 方法来创建辅缓冲区，代码如下所示：

```
int CreateSecondaryBuffer (·LPDIRECTSOUNDBUFFER * ppBuffer, int channels,
                           int secs, int samplerate, int bitspersample, DWORD flags)
{
    HRESULT hr;
    stringstream err;

    DSBUFFERDESC dsbd;
    WAVEFORMATEX wfx;

    ZeroMemory( &wfx, sizeof( WAVEFORMATEX) );
    wfx. wFormatTag = WAVE_FORMAT_PCM;
    wfx. nChannels = channels;
    wfx. nSamplesPerSec = samplerate;
    wfx. wBitsPerSample = bitspersample;
    wfx. nBlockAlign = ( WORD) ( wfx. wBitsPerSample / 8 * wfx. nChannels) ;
    wfx. nAvgBytesPerSec = wfx. nSamplesPerSec * wfx. nBlockAlign;
    ZeroMemory( &dsbd, sizeof( DSBUFFERDESC) );
    dsbd. dwSize = sizeof( DSBUFFERDESC) ;
    dsbd. dwFlags = flags;
    dsbd. dwBufferBytes = samplerate * bitspersample / 8 * channels * secs;
    dsbd. guid3DAlgorithm = GUID_NULL;
    dsbd. lpwfxFormat = &wfx;
    if( FAILED( hr = g_pDS-> CreateSoundBuffer( &dsbd, ppBuffer, NULL ) ) )
    {
```

```
        err << "CreateSecondaryBuffer: CreateSoundBuffer failed: " << DXGetErrorString8
        (hr)<<endl;
        throw(err.str());;
    }
    return(dsbd.dwBufferBytes); // 返回缓冲区的大小
}
```

代码的大部分与创建 DSBUFFERDESC 结构相关,这个结构告诉 DirectSound 想创建什么。DSBUFFERDESC 结构中的大部分变量的含义都是一目了然的。如果有不懂的,可以在 DirectX SDK 的帮助索引里输入 DSBUFFERDESC,得到关于每个变量的详细说明。

代码在创建缓冲区时使用了 WAVE_FORMAT_PCM 标志。这个标志告诉 DirectSound 在缓冲区中的数据格式用的是脉冲编码调制(PCM)。说到标准声效,如果没有指定压缩编码,那么指的就是 PCM。

创建 PCM 格式的缓冲区涉及到一些数学运算。比如,结构中的 block alignment 变量 nBlockAlign,它的运算看上去有些难于理解。在微软的文档上说,对于 PCM 格式的缓冲区,nBlockAlign 的值等于 nChannels 乘以 wBitsPerSample,再除以 8。对于 DirectSound 和声卡驱动程序中的一些运算,只需关心“它需要什么”,而不用关心“它为什么需要”。

提示

请注意创建缓冲区代码中的错误处理。应该用 FAILED 宏来检查 DirectSound 接口方法的返回值。如果出现错误,FAILED 宏将返回 true,以进行适当的操作。通常可以调用 DXGetErrorString8 函数来将错误代码转换成描述性的错误消息。

接下来看一下缓冲区描述结构中的变量 dwBufferBytes 的计算。通常,如果从硬盘读入 WAV 文件,就可以同时知道需要多大的空间,但程序则是自己产生声音数据的,因此需要计算所需存储空间。变量 samplerate、bitspersample、channels 和 secs 都是 CreateSecondaryBuffer 函数的参数。该函数将这些参数都乘起来,以决定总共需要的空间大小,然后把这个值赋给变量 dwBufferBytes。每一秒钟的音频有 samplerate 个采样点,因此总共有 secs × samplerate 个采样点。样本的声道数量存储在 channnels 中,而对于每一个声道都有 bitspersample 位,因此每个采样点的大小是 channels × bitspersample/8 字节。

像这样计算,将会得到一个长长的乘法表达式,这也是为什么 MP3 压缩很有用的原因。如果不进行压缩,一个立体声(双声道)的、CD 音质的(16 位采样,44 010 Hz 采样频率)、3 分钟(180 秒)长的歌曲所需要存储的信息量为 44 010 × (16/8) × 2 × 180 字节,也就是 31 687 200字节(大约为 31 MB)的存储空间。

下面来看一看程序代码,一旦创建了 DSBUFFERDESC 结构,程序代码就调用 IDirect-

Sound8 的 CreateSoundBuffer 函数来创建缓冲区,并将其接口放到 ppBuffer 中。注意,这里采用了双指针。变量 pBuffer 是指向 IDirectSoundBuffer 的指针(LPDIRECTSOUNDBUFFER 实际上是 IDirectSoundBuffer *,因此 LPDIRECTSOUNDBUFFER *实际上是 IDirectSoundBuffer **。根据命名法,LP 表示长指针,因此 LPDIRECTSOUNDBUFFER 是一个指向 DirectSound 缓冲区的长指针)。

提示

当进行每个样本的数学运算时,要细心。一个常见的错误是忘记除以 8,除以 8 使每样本多少位转变成每样本多少字节。如果计算缓冲区大小的时候忘记除以 8,将创建一个比实际需要的空间大 8 倍的缓冲区,这将浪费内存空间。如果在其他地方忘记除以 8,可能会发生更严重的错误。

请用长的、描述性的名字来命名变量。一个名为 bitspersample 的变量可以提示你需要除以 8,而一个名为 bps 的变量则会比较危险,因为你不知道"b"代表是位还是字节。

2.3.4 产生声音

创建了辅缓冲区之后,需要向它填充数据。在填充数据之前,一定要先锁定缓冲区(调用 IDirectSoundBuffer 的 Lock 函数)。Lock 函数告诉 DirectSound 将修改缓冲区的内容。当调用 Lock 函数时,将得到一个指向缓冲区的指针。如果 bitspersample 的值是 8,应该先把返回的指针类型转变成 unsigned char *,然后就可以填充数据了。在填充完成后,应该取消对缓冲区的锁定(调用 Unlock 函数)。下面是上述过程的程序代码:

```
void FillBuffer(LPDIRECTSOUNDBUFFER pBuffer, float frequency, int buffersize, int samplerate)
    {
        HRESULT hr;
        stringstream err;
        unsigned char * pBufferBytes;
        DWORD lockedsize;

        // 锁定缓冲区
        if(FAILED(hr = pBuffer -> Lock(0, buffersize, (void **)(&pBufferBytes),
                    &lockedsize, NULL, NULL, 0L)))
        {

        |
```

```
        err << "FillBuffer: Lock failed:" << DXGetErrorString8(hr) << endl;
        throw(err.str());
    }

    for(DWORD q=0; q<lockedsize; q++)
    {
        /*What cycle are we in? */
        float pos = frequency/(float)samplerate * (float)q;
        /*Take fractional part of cycle and convert to degrees */
        float r = (pos-floor(pos)) *2 * PI;
        float value = sin(r); //we now have a value somewhere between -1 and 1

        //change multiplier to change amplitude of wave aka volume
        pBufferBytes[q] = 127 + (value * 127);
    }

    //Unlock the buffer
    pBuffer -> Unlock( pBufferBytes, lockedsize, NULL, 0 );
}
```

这段代码从锁定缓冲区开始,调用 IDirectSoundBuffer 的 Lock 函数。它传递的参数有修改数据区的第一个字节(0)和修改数据区的大小(因为操作涉及整个缓冲区,所以大小为buffersize)。

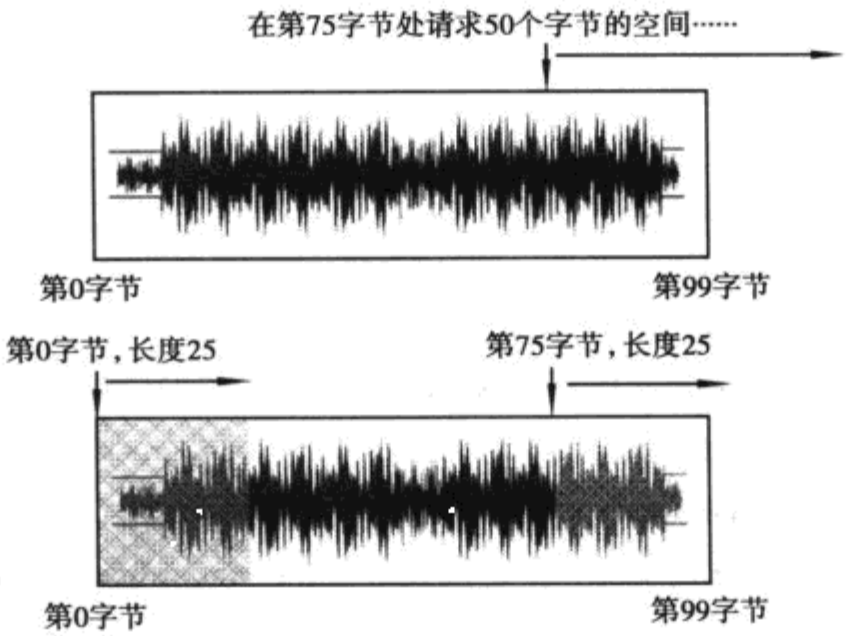
锁定操作返回的不是一个而是两个指针,这有助于使用流技术,参见图2.5。

DirectSound 允许指定开始字节和长度,且长度可以大于缓冲区的长度。例如,分配一个100字节的缓冲区,然后从第75字节开始请求50个字节的空间。你会觉得这样肯定会出问题,因为第75字节后面只有25个字节可用了。但实际上,DirectSound 采用的是"滚动"使用缓冲区。对一个100个字节长的缓冲区从第75字节开始请求50个字节时,会得到从 DirectSound 返回的两个指针:第一个指针从第75字节到第99字节,第二个指针从第0字节到第24字节。

由于自动地滚动使用缓冲区,DirectSound 不用做两次锁定/填充/取消锁定的操作。

如果确信被锁定的空间不会出现滚动,可以将第二个指针赋值为 NULL,并只使用 DirectSound 传递的第一个指针。音阶生成器示例程序中就使用了这种做法。当搬动数据时,程序传给 Lock 函数两个缓冲区和两个大小值。注意 Lock 函数的返回值,便可以很好地进

行滚动处理了。



……DirectX将返回两个指针，一个指向第75~99字节，一个指向第0~24字节。

图 2.5　锁定操作得到 2 个返回指针

提示

Lock 函数接收一对有用的标记。DSBLOCK_ENTIREBUFFER 告诉 Lock 函数锁定整个缓冲区。书中没有在示例程序中使用这个标记是为了明确地向读者展示如何锁定数据。但在通常的程序中，可以使用这个标记。

另外一个标记是 DSBLOCK_FROMWRITECURSOR，它告诉 DirectSound 你想从上次写操作结束的地方开始写数据。这有点像对文件的写操作，它在流技术中很有用。

一旦缓冲区被锁定后，实际操作就开始了。为了播放一个声音，示例程序向缓冲区写入了由正弦函数产生的数据。因为所做的都是三角函数的操作，所以正弦函数始终返回 -1 ~1之间的值。

这里，需要正确获得波的频率。完成这件事的算法要首先确定所处的周期，即此前经过了几个完整的波。当 pos 为一个整数时，表示开始一个新的波。图 2.6 形象地演示了这一个过程。

在程序代码中，表达式 pos-floor(pos) 表示了 pos 的“剩余部分”，也就是当前波经过了多少(该值始终在 0 ~1 之间)。比如，如果 pos 的值是 2.54(这意味着已经经过了两个波，且正经过第三个波的一半)，则 floor(pos) 的值为 2，r 的值为 0.54。

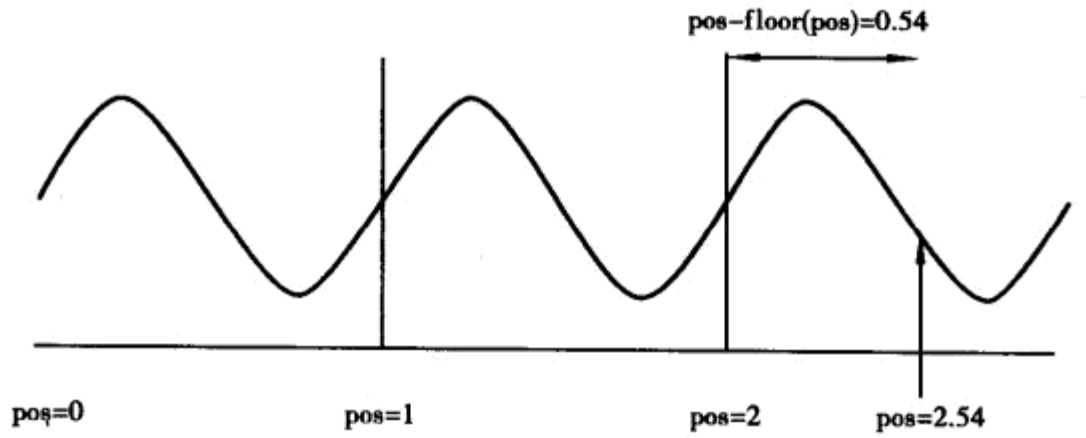


图2.6 用pos的小数部分来决定所处的位置

无论实际的波是长还是短，总能够找到一个介于0～1之间的剩余值。不用对频率做任何操作，因为在决定周期pos时，已经考虑了频率。

于是把r的值代入正弦函数中，以求得波在该点的振幅。这将得到一个介于－1～1之间的振幅。接着，将振幅乘以127。波的振幅表示音量，通过乘以127，表示设置音阶的最大音量。如果想让它只有一半的音量，可以乘以64。

现在波的振幅介于－127～127之间（如果你减半音量，则是－64～64之间）。最后一步是移动波值，使它的值符合无符号字节所表达的范围，即0～255。你只需简单的加上127就可以做到。

提示

音阶生成器也可用余弦函数来代替正弦函数。正弦函数的好处在于它从0开始。

如果做的是一个“实际的”音阶生成器，而不是示例程序，通常会选缓冲区大小为波的频率的整数倍。这样，缓冲区中的最后一个字节与第一个字节是无缝连接的。在示例程序中，根据所选频率的不同，缓冲区可能不是无缝地进行循环，因此在缓冲区循环播放时，可能会听到一个细微的嘀嗒声。

程序代码就是这样设置辅缓冲区中的每个字节。当for循环完成后，调用Unlock函数取消对缓冲区的锁定，调用时需要提供由Lock函数获得的指针和长度。Unlock不使用任何标记，它只是取消先前锁定的缓冲区。

提示

ToneGenerator示例程序不支持任务切换，如果想在游戏中实现这个功能，需要在任务切换回游戏时重新加载缓冲区，其操作可以通过IDirectSoundBuffer接口的Restore函数来完成。关于如何“精细地”重新加载，可以查看DirectX SDK和DirectSound示例程序。

2.3.5 播放缓冲区

经过创建和填充缓冲区的操作,会发现合成辅缓冲区与主缓冲区的信息并从扬声器发声是一件很简单的事情。所需要做的事就是调用对缓冲区操作的 Play 函数,其格式如下:

```
void PlayBuffer(LPDIRECTSOUNDBUFFER pBuffer)
{
  pBuffer->Play(0, 0, DSBPLAY_LOOPING);
}
```

Play 的第一个参数被保留,用于 DirectSound 的后续版本,且必须置零。第二个参数是指优先级,但只作用于特定的缓冲区(通过指定 DSBCAPS_LOCEDEFER 标记创建的缓冲区)。因为没有用这个标记创建辅缓冲区,所以只需置零而不用指定优先级。第三个参数是一个始终都有的标记参数。此外,如果创建的是一个 DSBCAPS_LOCEDEFER 的缓冲区,那么可以选择一些排序标记(详细的列表请参见 DirectX SDK 帮助)。但是,现在只创建了一个简单缓冲区,只可以使用 DSBPLAY_LOOPING 标记,这个标记告诉 DirectSound 无限次地循环使用指定缓冲区。

使用 Stop 函数停止播放声音。Stop 函数没有参数,它所做的事是停止播放当前正在播放的缓冲区。

2.3.6 清理资源

当用户听到一个音阶并按键停止播放时,程序代码便开始清理先前分配的资源:

```
// 释放辅缓冲区
cout << "Releasing sound buffer..." << endl;
SAFE_RELEASE(soundbuf);

// 清除对 DirectSound 的初始化
cout << "Un-Initializing Audio..." << endl;
SAFE_RELEASE(g_pDS);
```

示例程序只分配了两个资源:辅缓冲区和 DirectSound 接口(该接口也包含主缓冲区)。下面是 SAFE_RELEASE 的定义,它是在 dxutil.h 文件中声明的:

```
#define SAFE_RELEASE (p)      { if (p) { (p)->Release(); (p)= NULL; } }
```

SAFE_RELEASE 检查传给它的接口,如果接口不为 NULL,则释放它(调用 Release),然

后将它设置为 NULL(否则,当你想再使用它时,会出现访问失败,并且你可能不会知道错误原因)。这是 DirectX 的标准,最好使用 SAFE_RELEASE,而不直接调用 Release 函数。

此外,要确保以正确的次序释放资源,应该在释放主 DirectSound 接口之前释放所有的辅缓冲区。否则,DirectSound 将警告用户忘记释放辅缓冲区,因为它认为在被释放时,凡没有被释放的缓冲区都是被忽略了。

2.4 本章小结

希望读者能喜欢这段关于 DirectSound 的简短学习,在学习中,本章给出了一个音阶生成器的示例程序。

在实际游戏中,可能永远不会使用音阶生成器。后面章节中,读者将会看到 DirectX 提供的很多高层控制操作,它使你不用自己创建音阶来编制音乐。如果你正在编制一个游戏,并且希望能有很酷的音乐和声效,可以使用本章中的代码来创建一个合成器引擎,但这要重新编写许多功能函数。在接下来的章节中,将会学习有哪些功能函数已经存在以及如何使用它们。

本章的示例程序有两个目的:第一个目的是说明初始化和使用 DirectSound 的基本操作;第二个目的是说明当播放声音时低层进行了什么操作。当锁定了缓冲区,以字节的形式处理音频的时候,你就处在低层操作上。希望通过本章的示例程序你已经知道低层实际发生的事情。如果理解了某件事情的核心操作,那么很容易理解较高层次的操作。

现在,我们完成了第一个 DirectSound 程序。接下来,将学习如何利用 DirectX Audio 较高层次接口来完成其他的声效和音乐任务。



PDG

第 3 章

播放 WAV 文件

前两章所讨论的基础知识的目的是帮助读者较好地理解以后章节的内容，但在实际游戏编程中并不实用。从本章开始，将学习一些实用技术。这些技术可以直接应用到游戏编程中。

本章开始的示例程序将创建一个 C ++ 音频引擎，后面的章节都将围绕这个引擎的开发而展开。学完这几章后，将获得一个功能完备的音频引擎，可以把它放入自己的项目中，也可以作为自己设计引擎的起点。

首先，需要理解几个新的概念，包括令程序员头疼的 DirectMusic。

3.1 用 DirectMusic 处理声效

对于大多数的游戏编程员，DirectMusic 是一个令人费解的代码块。以前它和 DirectX 捆绑在一起，虽然微软已经作了一些简化，但是 DirectMusic 依然较复杂。

DirectMusic 是一个包罗万象的类库，它因为庞大而显得复杂。它可以播放 WAV 文件，可以用流技术从磁盘上处理声波，可以播放 MIDI 文件，可以播放动态音乐，甚至可以用于作曲。几乎没有 DirectMusic 不能做的事情。

DirectMusic 功能很强，但同时也难懂。有句行话："大的类库使复杂的事情变得简单，也使简单的事情变得复杂"。例如，将 WAV 文件从硬盘上取出变成声波流是很容易的事情：对它加载，获得一个段接口，告诉执行代码播放这个段。在 DirectSound 中，把 WAV 文件变成声波流相对复杂得多（本章的后面部分将会提到）。这是一个"复杂变简单"的例子。

另一个例子是一件令我难忘的事情，它是关于音量控制的问题。我花了很长时间寻找 DirectMusic 的音量控制按钮，而当我最终找到它时，我发现有 6 种不同的方法可以控制音量，而如果其中一种方法关掉了音量，就将听不到声音。这使简单事情变得复杂了。

建议读者循序渐进地学习 DirectMusic，确保在学习下一个内容前已经真正理解了前一个内容。DirectMusic 虽然是一个庞大的 API 库，但掌握它并不难，它只涉及一些新的概念，并不涉及很复杂的数学运算。这跟 DirectGraphics 不一样，在 DirectGraphics 中如果不知道线性代数，就如同用轮胎漏气的汽车参加比赛一样。

本书将用几个章节来讲解 DirectMusic。本章先介绍一些核心概念，使读者可以用 DirectMusic 处理声效。在后续章节中，学习将逐步深入：第 5 章"控制行为"将介绍用于音乐

播放的接口,本书的第二部分将介绍关于动态音乐的接口。

下面将按照图3.1所示的流程逐项进行学习。

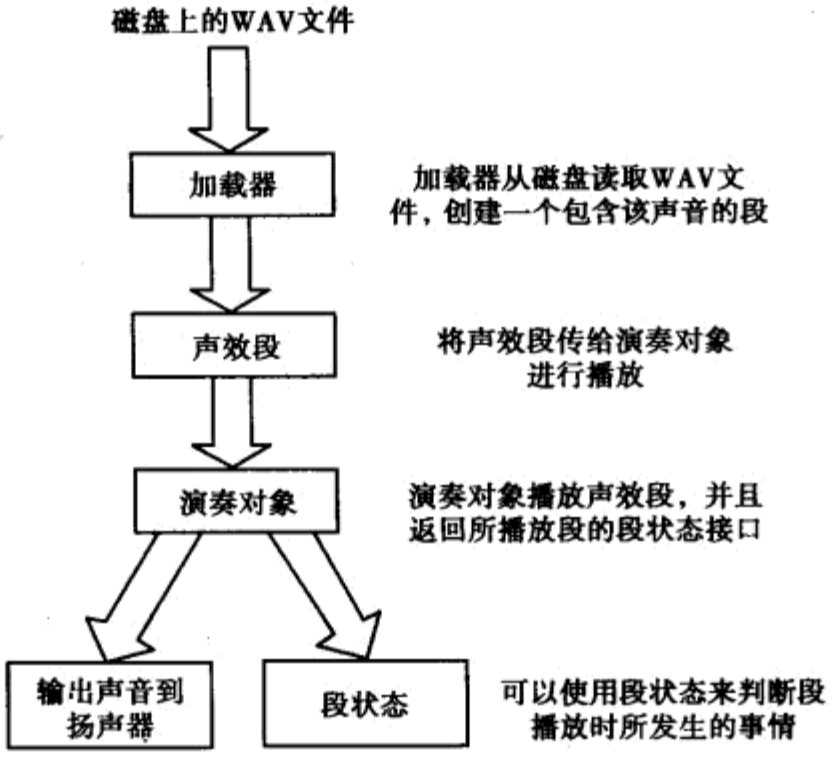


图3.1 DirectMusic播放声效的流程及接口

3.1.1 加载器

首先从读取硬盘上的WAV文件开始。在DirectMusic中,通过一个特定的加载器来完成加载。可以通过IDirectMusicLoader8接口访问这个加载器。

到目前为止,加载器位于所有DirectMusic接口的最低层,加载WAV文件是一项复杂的任务。加载器处理加载WAV文件所涉及具体操作为:判断文件是否被压缩和如何压缩,然后读取字节并存放在内存中。

加载器的功能很强,可以告诉它在哪儿查找支持文件,可以让它读取依附于EXE程序的资源,甚至可以编写一个自定义的加载器对象,教它如何加载自己的声音格式。

3.1.2 声效段

当加载器加载某个文件时,加载器会创建一个DirectMusic声效段,并把从驱动器上读取的数据放入该段中。可以通过IDirectMusicSegment8接口来访问DirectMusic的声效段。

一个声效段可以包含任何一个音频段,音频段包括WAV文件和任意长度的音乐。每个段可以包含多个音轨,可以把音轨看成是音乐中一条单一的谱线,音轨可以通过IDirectMusicTrack8接口播放。例如一首钢琴曲谱,有两个音轨,上面的音轨是用右手弹奏的(通常是高音谱号),另一个是用左手弹奏的;而一首典型的歌曲可能包括重低音音轨、打击乐音

轨、弦乐音轨和人声音轨(或是主旋律音轨)。当加载一个 WAV 文件并放入一个声效段时,是用一个音轨来结束一个段的。虽然通常不用直接处理音轨,但是最好知道它们的存在。

有多种方法来创建声效段,而且都可以通过类库自动完成。比如,如果要使用动态音乐,可以让 DirectMusic 来创建声效段。只需提供弦乐、乐器和主旋律,DirectMusic 就创建出相应的声效段。但请注意不要让播放器一遍又一遍地重复播放一首歌曲。

可能跟读者期望的不一样,IDirectMusicSegment8 接口并没有提供 Play 函数,不能直接调用函数来播放声效段。为了能听到声音,需要使用另外一个“演奏对象”。

3.1.3 演奏对象

演奏对象是控制其他对象的主对象。一旦获得了 IDirectMusicPerformance8 接口,就可以用它来播放声效段,控制哪些段由哪个扬声器播放,而且还可以调节速度、音量和其他影响声效段播放的设置。

简单地讲,播放一个声效段所要做的事情就是向演奏对象发出指令,让其播放这个声效段。演奏对象的时间控制是可以查询和设置的。例如,可以查询演奏对象当前的延时是多少(指从开始“播放”到扬声器实际发声之间的间隔)。

3.1.4 其他对象

如前所述,DirectMusic 有很多其他对象和接口,但读者只需要知道刚学过的 3 个接口对象:加载器、声效段和演奏对象。其他一些与 DirectMusic 播放音乐相关的对象,将在第 5 章学习。

现在已经介绍了本节流程中涉及的几个接口的功能,在学习如何使用它们来构建音频引擎前,还需要了解如何处理错误。

3.2 音频引擎中错误处理

设计一个引擎最重要的一个特点是鲁棒性,或在极端环境下不死机的能力。确保代码执行时不死机非常重要,涉及到引擎代码时就更为重要。我敢断言:当编程者编好引擎并开始使用它开发游戏时,已经忘记了它的复杂程度。在编制游戏的过程中,没有必要重新了解一个 API 或者引擎是如何工作的。因此,需要预先花点时间考虑加入一些可靠的错误处理。

错误处理的方法是使用 C ++ 的异常处理。如果以前没有用过异常处理,应该先去学习它,然后再继续后面的学习。现在,本书假定读者已知道 C ++ 的 try、throw 和 catch 语句。如果这些语句对你来说比较陌生,可以通过配套光盘提供的链接学习相关的内容。

3.2.1 是否需要异常处理

笔者认识很多程序员,他们对异常处理有不同的看法。有些人认为这是编写自动防止错误代码的惟一方法;另一些人则认为异常处理的工作机制不好,它有 goto 语句并且优化不成熟。支持者认为异常处理使代码更加具有鲁棒性,并且更容易维护;反对者则断言它将会产生臃肿的代码和松散的编程。对于异常处理,本人的立场是倾向于支持提供异常处理能力。

3.2.2 异常处理的规则

作者个人认为异常处理可以使代码更容易阅读和书写,它需要遵循几个重要的规则:

①如果抛出异常,则需要捕获异常。

②抛出异常是在某些不期望发生的事情发生时,而不是在某些不好的事情发生时。

③尽可能早的捕捉错误,但不一定是在最近的时刻。

下面,逐项解释这些规则。

规则1:如果抛出异常,则需要捕获异常

换句话说,当出现错误时,就要决定需要捕获哪些错误以及这些错误是什么。

如果抛出一个错误没有被程序捕获,则称为“没有处理的异常”,它将挂起程序并出现类似于图3.2的错误对话框。很多程序因错误编码而引起死机,而错误的原因仅仅是一个相对无害的异常“溜了出来”,没有被捕获。

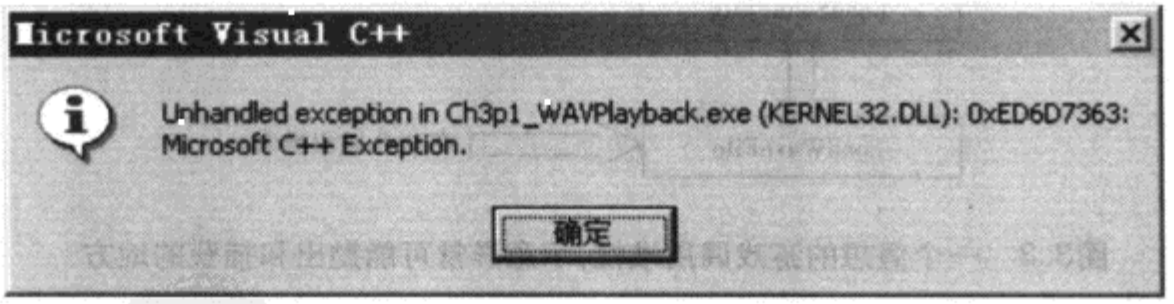


图3.2 当没有捕获异常时所出现的对话框

在使用没有异常处理的大型代码库时,这条规则特别重要。

规则2:抛出异常是在某些不期望发生的事情发生时,而不是在某些不好的事情发生时

关于这条规则有一个经典例子:某人正在尝试登录一台服务器,有两条“正常的”执行路线:或者是服务器允许他们登录,或者是服务器拒绝他们的访问。这两条路线的编码不需要异常处理,比如,返回“真”值表示可以访问,返回“假”值表示不能访问。

一种异常的情形是服务器返回无用信息:它既不说“是”,也不说“否”,而说一些没有多

大意义的术语。这时,抛出一个异常是恰当的。因为这种情形不正常,表示用户遇到了比密码输入错误更大的麻烦。

另一种异常的情形是客户端不能访问服务器,或者是因为内存耗尽,或者是因为不能通过网络发送数据。

为什么要遵守这条规则呢? 起初,异常处理非常慢。编写 C ++ 编译器的人假设前提是:只会在情形很糟糕的时候程序才会抛出异常,在这种情况下,没有谁会关心它是否运行缓慢,至少它避免了死机。无论什么原因,抛出异常确实会减慢处理速度。另一个原因是频繁使用异常会使代码难于阅读。

规则 3:尽可能早地捕捉错误,但不一定是在最近的时刻

显然,谁都不期望在没有能力处理异常时捕获异常。处理错误的代码应该位于命令链中足够高的地方,以便它能够决定当错误发生时做什么事情。

假定正在编写读取一个 WAV 文件并把它存放在 CSound 类中的程序代码。这个操作会用到几个函数,如图 3.3 所示:在顶端是 C ++ 程序的 main 函数,main 函数调用 LoadGameResources 函数,这个函数负责为游戏加载图画、音乐、游戏难度。

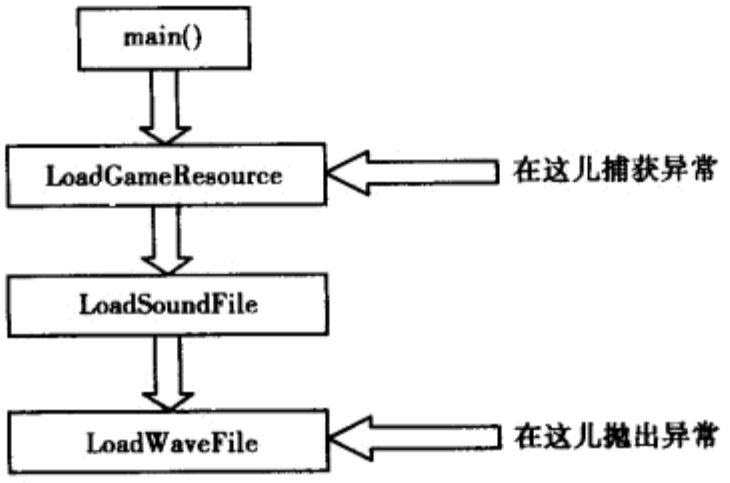


图3.3 一个假想的游戏调用堆栈,示意异常可能抛出和捕获的地方

随后,LoadGameResources 函数调用 LoadSoundFile 函数,而 LoadSoundFile 函数又调用 LoadWaveFile 函数来加载 WAV 文件,调用 LoadMP3File 函数来加载 MP3 文件。

假设 LoadWaveFile 函数执行 fopen 命令打开硬盘上的 WAV 文件,而 fopen 执行失败(假设要打开的文件不存在)。这时使用异常处理,当 fopen 执行失败时,会抛出一个错误。

规则 1 说明如果抛出异常,则需要捕获异常,因此现在需要决定在哪儿捕获异常。有好几个地方可以捕获异常:

- 在 LoadWaveFile 函数中
- 在 LoadSoundFile 函数中
- 在 LoadGameResources 函数中

- 在 main 函数中

从 LoadWaveFile 函数开始考虑。在这个函数的异常处理中可以做什么事情呢？因为 LoadWaveFile 函数不知道文件被加载的原因，因此它不能决定当有错误发生时该做什么。其反例是 LoadWaveFile 函数捕获错误，并调用 exit(-1)退出。对于游戏而言，这样做可能是可行的，但如果在一个编辑音频的应用程序中这样使用 LoadWaveFile 函数，就会使应用程序因为一个需要加载的文件没有找到而立即退出，从而丢失用户所有数据。一个较好的方法是弹出一个“文件没有找到”的消息框，并且允许用户继续使用应用程序。LoadWaveFile 函数仅仅是一个“子函数”，它所涉及的程序上下文信息比较少。

往调用堆栈上面走，LoadSoundFile 函数如何呢？LoadSoundFile 函数也不是捕获异常最好的地方。它与 LoadWaveFile 函数一样缺乏上下文信息。

下一个是 LoadGameResources 函数。这是第一个知道足够多上下文信息来判断需要做什么的地方。LoadGameResources 函数知道如果不能从驱动器中读取数据并放入 RAM 中，游戏将不能按照设计的要求正常工作。在这个函数中，可以编写代码来弹出一个消息框，告诉玩家重新安装游戏，或者尝试修复这个错误(可以检查光驱里有没有游戏 CD，如果有，就从光盘里加载剩余的资源)。

最后看看在 main 函数里捕获错误又会怎么样？当然可以在 main 函数里处理错误，但 main 函数位于调用堆栈的顶端，它决定函数调用，当然不是捕获错误的最佳地方。作者认为，用 main 函数处理错误就如同 Picard 把 Geordi 赶出轮机舱，自己修理故障一样。因此，似乎最好由 LoadGameResources 函数捕获错误，并尝试修复错误。但依旧有问题，这时 main 函数会捕获到重新抛出的错误。这就如同一个主管操作的领导告诉他们公司的 CEO，“我们现在有这样一个问题，我尝试解决它，但问题对于我来说太大了，我需要你的帮助来继续工作”。

确定捕获错误的最佳地方必须通过积累实践经验才能正确把握。这里仅给出一些指南，不一定适用所有情况。一般地，在“较低的地方”(更接近错误抛出的地方)捕获错误比“较高的地方”捕获错误更糟糕，特别是对于游戏编程。游戏编程中异常错误经常出现，因此在较高的地方捕获它们是可行的。即在 main 函数里捕获所有错误，然后弹出一个消息框提示“游戏退出”，并清理占用资源，这种做法比较好。现在，市面上有些游戏经常因为一个非法访问而引起停机，并让系统处于完全混乱的状态。

此外，在播放时存在报酬递减规律：针对错误做“完全正确的事”常常需要大量的代码，但却得不到多少好处。除非编制的是生命支持软件或火箭指挥系统，否则对于游戏编程而言，不必考虑让游戏十全十美。

3.3 设计音频引擎

前面已经学习了相关概念,现在开始学习如何设计一个音频引擎。请查看 Ch3p1_WAVPlayback 示例程序,它使用我们初现雏形的音频引擎来播放一个 WAV 文件。

3.3.1 音频引擎中异常处理

前面学习了异常处理的规则,在本节中,将利用所学知识来创建引擎的错误处理逻辑。将要讲述的代码在 ErrorHandling. h 和 ErrorHandling. cpp 源文件中。

(1) CError 对象类

所有工作都从 CError 对象开始。在很多时候,当程序员使用异常时,他们不抛出诸如字符串或原始数据类型错误。取而代之,他们抛出异常类,这些类不仅包含了错误,而且包含一些具有诊断性的信息和关于错误发生的原因和时间的上下文信息。本书中的音频引擎将这些信息存放在 CError 对象中。这个对象的代码如下:

```
class CError
{
    public:

    CError(HRESULT hr, std::string err, std::string filename, int line) {
       SetFile(filename);
       SetError(err);
       SetLine(line);
       if (hr) SetReason(DXGetErrorString8(hr));
}
   virtual ~CError() { }

   std::string GetFile() { return(m_File); }
   void SetFile(std::string f) { m_File = f; }

   std::string GetError() { return(m_Error); }
   void SetError(std::string f) { m_Error = f; }
```

```
    int GetLine( ) { return(m_Line); }
    void SetLine(int l) { m_Line = l; }

    std::string GetReason( ) { return(m_Reason); }
    void SetReason(std::string f) { m_Reason = f; }

    std::string GetMessageBoxString( );

    protected:
    std::string m_File;
    int m_Line;
    std::string m_Error;
    std::string m_Reason;
};
```

正如所看到的,CError 是一个相当简单的类。它保存了产生错误的源文件的文件名(m_File),在源文件中的行号(m_Line),显示给用户看的错误信息字符串(m_Error)和一个内部可诊断的消息(m_Reason)。它有关于所有成员变量的访问方法,一个可以按需自动调用 DXGetErrorString8 方法的简短的构造函数,以及一个将变量格式化成用户显示文本消息的 GetMessageBoxString 方法。

(2)错误处理宏

为了让 CError 的处理相对容易一些,笔者曾写了一些用#define 定义的宏。#define 代码基于 Steve Rabin 的文章《从断言中获取更多的东西》(《Squeezing More Out of Assert》),这篇文章收录在 Game Programming Gems 1 一书中,书号为 ISBN 1-58450-049-2。我向所有的游戏编程员推荐这本书(并非是因为我为它写了一本姐妹书)。在 ErrorHandling.h 文件中,有两个宏,分别是 ThrowIfFailed 和 Throw。特别注意用大写字母 T,以区别关键字 throw。先讲解 Throw:

```
#define Throw(err) { \
    if (!ThrowCError(NULL, err, _FILE_, _LINE_)) { \
    _asm { int 3 } \
    } \
} \
```

Throw 宏是调用 ThrowCError 全局函数的缩写。CError 通过宏获得 m_File 和 m_Line 变

量,宏自动提供了当前的文件名和行号,这里引用了内部的_FILE_和_LINE_宏。

ThrowCError 函数负责创建 CError 对象:

```
bool ThrowCError(HRESULT hr, std::string err, std::string filename, int line)
{
  CError e(hr, err, filename, line);
  string displaystr = e.GetMessageBoxString( );
  displaystr += "Press ABORT to end the program, RETRY to debug, IGNORE to throw the
  error.";
  int result = MessageBox(NULL, displaystr.c_str( ),
    "ErrorHandling.cpp", MB_ABORTRETRYIGNORE | MB_ICONSTOP);
  switch(result) {
  case IDABORT:  // 立即结束程序
    exit(-1);      // 在这儿也可以使用 abort( )函数
  case IDRETRY:  // 跳入调试器
    return(false); // 让宏来做,因此我们停在实际的出错行上
  case IDIGNORE: // 继续正常执行(抛出错误)
    throw(e);
  }
  return(true);        // 只是为了避免编译器的警告
}
```

ThrowCError 函数的任务就是弹出一个有 Abort、Retry 和 Ignore 按钮的消息框(见图 3.4)。如果用户选择 Abort 按钮,函数立即退出。如果选择 Ignore 按钮,代码抛出错误,就好像消息框从来没有显示过。如果选择 Retry 按钮,函数返回假值。再看一下 Throw 宏的定义——如果 ThrowCError 函数返回假值,我们执行(汇编状态的)int 3 指令。int 3 指令发出调试中断。当程序在调试器中运行时,调试中断会立即中断程序执行。

为什么不把 int 3 指令直接放在 ThrowCError 函数里呢? 因为如果把 int 3 指令作为 ThrowCError 函数的一部分,在点击"重试"按钮时,就将中断进入 ThrowCError 函数的代码,则不得不向上移动堆栈来看实际抛出了什么错误。而把 int 3 指令作为宏的一部分,在中断时,可以立即转到源文件出错的地方。这是一个技巧,也是我赞赏 Steve Rabin 的原因之一。

理解了 Throw 宏之后,再看一下更复杂的 ThrowIfFailed 宏:

```
#define ThrowIfFailed(result, err) { \
  if(FAILED(result)) { \
    if (!ThrowCError(result, err, _FILE_, _LINE_)) { \
```

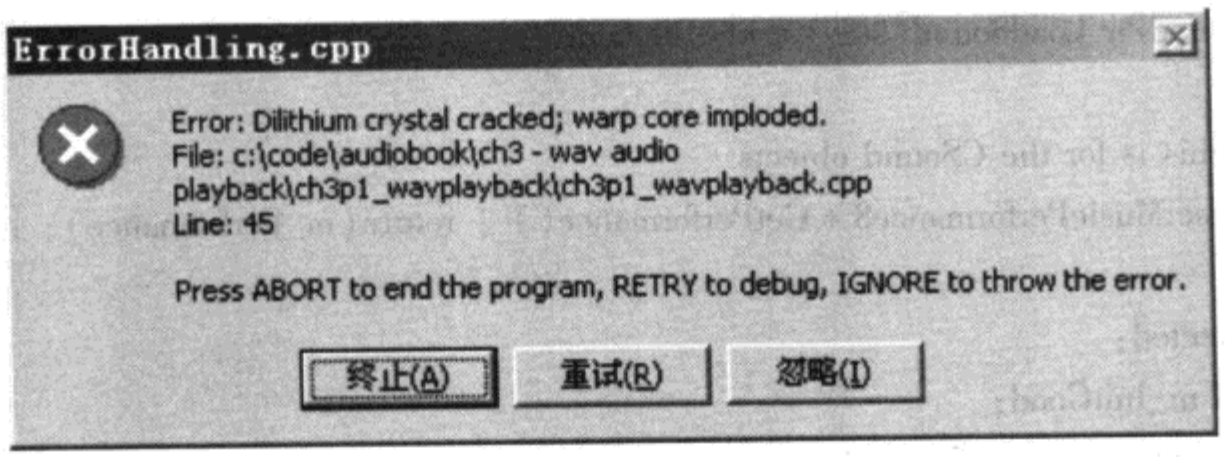


图3.4 异常处理代码所产生的消息框

```
            _asm { int 3 } \
        } \
    } \
} \
```

当传递的HRESULT参数是一个错误代码时,这个宏与Throw宏所做的事情是一样的。如果FAILED(result)返回真值,则代码将HRESULT传递给ThrowCError函数,而ThrowCError函数又传递给CError的构造函数。构造函数查询错误代码并将与之对应的错误字符串放入m_Reason中。

这两个宏很方便,因此将看到它们会在整个音频引擎中像类库一样出现。总之,这个小小的错误处理系统使代码具有鲁棒性,当然,还可以加入更多的东西。当出现错误时,这个简单的模式会带来真正的帮助。

3.3.2 音频管理器对象类

异常处理已经完成,现在可以开始编写管理整个游戏音频的CAudioManager类了。毫无例外,任何功能强大的引擎都是从一个简单的头文件开始的。下面是关于CAudioManager类的声明部分:

```
class CAudioManager
{
    public:
    CAudioManager();
    virtual ~CAudioManager();

    void Init(HWND hwnd, bool stereo = true, int perfchannels = 128);
    void UnInit();
```

```
    CSoundPtr LoadSound(std::string filename);

    // this is for the CSound objects
    IDirectMusicPerformance8 * GetPerformance() { return(m_Performance); }

    protected:
    bool m_InitGood;
    IDirectMusicLoader8 * m_Loader;
    IDirectMusicPerformance8 * m_Performance;
    static const int CAudioManager::m_PerfChannels;
}
```

显然,这只是一个开始,最终完成引擎还会用到很多方法。例如初始化引擎、反初始化引擎以及加载声音等方法,也都能直接获得执行接口。CAudioManager 类的内部包括执行接口和加载器接口,并且用 m_InitGood 来标记管理器是否被正确地初始化。m_PerfChannels 常量表示执行通道的数量,可直接理解为可以同时播放声音的数量。

提示

CD 光盘里包含了多个音频引擎的源代码映像,分别代表某一章。也就是说,在第 3 章目录中的 AudioManager. cpp 和 AudioManager. h 文件只包含第 3 章所讨论的功能。在第 4 章目录中的文件包含本章的内容,外加第 4 章所增强的部分。这使得读者可以随着本书的引导添加功能,而通过所提供的多个不同的历史版本,读者也可以用自己的方式开始编写代码。

如果想获得并使用本书中的音频引擎,请使用光盘中 AudioEngine 文件夹中的音频引擎,它拥有所有的功能。

(1)初始化

CaudioManager 使用“两阶段创建”。编程术语“两阶段创建”的含义是创建一个对象不仅需要调用构造函数,而且要调用初始化函数。本书把这个初始化函数称作 Init 函数,也有程序员把它称作 Create、Setup、Initialize 或者 Reset。

两阶段创建是一个好主意,有两个原因:第一,它适合于控制台程序或低内存平台,像 PDA。在这些设备上,需尽量减少游戏分配内存的次数,因此,当程序启动时,可以使用两阶段创建来初始化游戏需要的每一个对象。一旦游戏开始,就不允许再调用 new 函数,取而代之,调用对象的 Init 函数,该函数使对象变成当前状态。这有点像得到一个新的对象,但

又不用进行繁琐的内存分配。

本书的音频引擎实际上并没有考虑上述的情况,但它确实考虑了使用两阶段创建的第二个原因:在对象创建后,对其进行控制。为了使引擎尽可能的灵活,要求对象在初始化以后能够被控制。因此,引擎使用了两阶段创建。

初始化 Init 函数的代码如下:

```
void CAudioManager::Init(HWND hwnd, bool stereo, int perfchannels)
{
    HRESULT hr;

    // 初始化 COM
    hr = CoInitialize(NULL);
    ThrowIfFailed(hr, "CAudioManager::Init: CoInitialize failed.");

    // 创建加载器
    hr = CoCreateInstance(CLSID_DirectMusicLoader, NULL, CLSCTX_INPROC,
      IID_IDirectMusicLoader8, (void **)&m_Loader);
    ThrowIfFailed(hr, "CAudioManager::Init: CoCreateInstance for loader failed.");

    // 创建演奏对象
    hr = CoCreateInstance(CLSID_DirectMusicPerformance, NULL, CLSCTX_INPROC,
      IID_IDirectMusicPerformance8, (void **)&m_Performance);
    ThrowIfFailed(hr, "CAudioManager::Init:CoCreateInstance for performance failed.");

    hr = m_Performance -> InitAudio(NULL, NULL, hwnd, stereo ?
      DMUS_APATH_DYNAMIC_STEREO: DMUS_APATH_DYNAMIC_MONO,
      perfchannels, DMUS_AUDIOF_ALL, NULL);
    if (hr == DSERR_NODRIVER) {
      // 输入一个警告信息,然后继续执行
      MessageBox(hwnd, "The program could not locate your audio hardware.", MB_ICON-
      STOP);
      return; // 注意我们没有设置 m_InitGood 为真值
    }
    else ThrowIfFailed(hr, "CAudioManager::Init:m_Performance ->InitAudio failed.");
    m_InitGood = true;
```

函数的3个参数中有两个是可选的,只有窗口句柄是必须的。如果需要,也可以指定执行的音频是立体声还是单声道,以及最大通道数。通道的开销是比较小的(除非它们被播放),因此设置的通道数量可以大于实际所需要的。通道数量的默认值是128。

代码从初始化 COM 开始,然后获取一个 DirectMusic 加载器的接口。随后得到一个演奏对象。所有这些调用都使用 CoCreateInstance,这是一个 COM API 函数,可以用它来创建任何一个 COM 对象。其他组件将 CoCreateInstance 隐藏在函数内部,而在 DirectMusic 中有很多 COM 接口,因此不得不直接使用 CoCreateInstance 函数。CoCreateInstance 函数实质上是创建某个类的对象和能与该新对象交互的接口。CoCreateInstance 函数的参数从左至右分别是:类 ID(GUID 通常用 CLSID_开头的名称命名),它标示了创建的对象;集合指针(通常为 NULL);上下文(通常是 CLSCTX_INPROC,意思是新对象共享相同的地址空间作为创建器代码,就像 DLL);接口 ID(GUID 通常用 IID_开头的名称命名),它标示了要使用的与新建对象交互的接口;最后一个参数是指向新接口的指针的地址。代码创建完所有对象以后,就调用刚创建的 IDirectMusicPerformance8 接口的 InitAudio 函数。InitAudio 函数初始化演奏对象,并创建一个默认的音频路径。代码检查 stereo 参数,并指定是 DMUS_APATH_DYNAMIC_STEREO 还是 DMUS_APATH_DYNAMIC_MONO。需要指出的是还有另外两个选项:DMUS_APATH_DYNAMIC_3D(用于创建3D声音)和 DUS_APATH_SHARED_STEREOPLUSREVERB(常被用于使用混响来增强音乐效果)。

注意代码检查 InitAudio 函数的返回值是否为 DSERR_NODRIVER。这是一个特定的错误,它指出 DirectX Audio 找不到任何音频硬件。当出现 DSERR_NODRIVER 错误时,代码弹出一个消息框告诉玩家没有找到他的音频硬件。

如果操作成功,即没有抛出错误,函数在结束时就设置 m_InitGood 标记为真,这使得后续的函数知道管理器是否能正常播放音频。

(2)反初始化

CAudioManager 类的 UnInit 函数描述如下:

```
void CAudioManager::UnInit()
{
    if(m_Performance!=NULL) {
    m_Performance->Stop(NULL, NULL, 0, 0);
    m_Performance->CloseDown();
    SAFE_RELEASE(m_Performance);
    }
```

```
    SAFE_RELEASE(m_Loader);
    m_InitGood = false;
}
```

代码从调用 m_Performance 的 Stop 和 CloseDown 函数开始，假定 m_Performance 非空，然后释放 m_Performance 对象和 m_Loader 对象，并清除 m_InitGood 标记。

(3)加载声音

到目前为止，CAudioManager 类中最重要的函数是加载 WAV 文件并由此创建一个音频对象，下面是相应的代码：

```
CSoundPtr CAudioManager::LoadSound(std::string filename)
{
    HRESULT hr;
    CSound * snd = new CSound(this);

    // 转换文件名为 wide-string
    WCHAR widefilename[MAX_PATH];
    DXUtil_ConvertGenericStringToWideCch(widefilename, filename.c_str(), MAX_
    PATH);

    // 让加载器加载这个文件
    hr = m_Loader -> LoadObjectFromFile(CLSID_DirectMusicSegment,
      IID_IDirectMusicSegment8, widefilename, (void **) &snd -> m_Segment);
    ThrowIfFailed(hr," CAudioManager::LoadSound: LoadObjectFromFile failed. ");
    return(CSoundPtr(snd));
}
```

> **提示**
>
> 把 WAV 文件当作是音乐的一个音符段，它被一个乐器演奏。你会得到一个声效段的接口，而它实际上是一个 WAV 文件。

DirectMusic 加载器加载 WAV 文件的过程分为两个阶段。首先，它把给定的文件名转换成宽字符串形式(DirectMusic 只能操作宽字符串)。然后，它让加载器调用 LoadObjectFromFile 函数加载 WAV 文件。它向加载器传递类ID(CLSID_DirectMusicSegment)、期望返回的接口 ID(IID_IDirectMusicSegment8)以及加载器所要求的参数。

代码把得到的声效段接口存放在 CSound 类的 m_Segment 变量中。(CAudioManager 类

是 CSound 类的友类,因此,它可以访问 CSound 类中受保护的成员变量。)

在下一节中将学习最后一行代码,现在暂且将这个返回语句记下来。

3.3.3 智能指针对象类

除了鲁棒性和错误处理以外,引擎需要实现的核心功能是资源管理。一个好的引擎应该让它的客户端比较容易创建对象、使用对象以及在不再使用时清除对象。

一种好的解决方法是使用智能指针(smart pointer)。智能指针也是一个对象类,它的行为跟普通的指针一样,惟一区别在于它是智能的,它知道什么时候清除不再使用的对象。

有些程序员使用智能指针来减轻内存管理的负担,特别是在使用异常处理时。如果想进一步了解什么是智能指针,请查阅光盘上的链接。

音频引擎里的智能指针都是从一个公共的基类 CRefCountPtr 继承下来的,这个基类也是一个模板,它是在 RefCountPtr. h 文件中实现的,它基本上采用 David Harvey 编写的代码。配套光盘中提供了原始代码文档的链接,如果想知道智能指针是如何工作的,可以到那里去学习。

CSoundPtr 是 CSound 指针的子类。本书提供了一个构造函数和一个 Release 函数,该函数输出消息到调试控制台,因此读者可以看到声音被实时地创建和清除的过程。

3.3.4 音频对象类

在音频引擎中,CSound 对象具体负责声效。这个对象的核心是保存 WAV 文件的 DirectMusic 段。CSound 类封装了声效段的接口。

不完全照搬 DirectMusic 的做法,我在 CSound 类中进行播放时,是直接向音频对象发出播放指令。另一种做法是把 Play 函数加入 CAudioManager 类中,并把 CSound 作为参数传递。

3.3.5 播放音频

将 Play 函数放到 CSound 类中,需要开放执行接口。这就是 CAudioManger 类有一个 GetPerformance 接口的原因。CSound 对象类使用这个方法,然后传递它自己的 m_Segment 参数来调用 PlaySegment 函数,其代码如下:

```
bool CSound::Play()
{
    if (NULL == m_Segment) return(false);
    if (NULL == m_Manager) return(false);
    m_Segment -> Download( m_Manager -> GetPerformance( ) );
```

```
    m_Manager -> GetPerformance( ) -> PlaySegment( m_Segment,0,0,( IDirectMusicSeg-
    mentState ** ) &m_SegmentState) ;
    return( true) ;
}
```

这个函数的核心是调用 PlaySegment 函数。为了播放一段声效,必须向执行接口提供要播放的声效段(m_Segment)、所需标记(这儿没有标记)和开始播放的时刻(0,“尽可能的早”)。这个函数返回一个 IDirectMusicSegmentState 接口,这个接口可以用来查询正在播放的声效段。

3.3.6 下载音频

请注意在 Play 函数中调用 PlaySegment 函数前的一行代码,播放音频前需要下载合成器。如果不这样做,将不能播放音频。通过调用 Download 函数来下载一个声效段,调用时,将用来播放的执行接口作为传递参数。

提示

下载声效段麻烦吗? 你可以让 DirectMusic 在需要时自动下载。通过调用 IDirectMusicPerformance8 接口的 SetglobalParam 函数来完成,调用时传递参数 GUID_PerfAutoDownload。请查看 DirectX 文档以获得更详细的资料。

3.3.7 检测音频播放状态

可以根据 PlaySegment 函数返回的 IDirectMusicSegmentState 接口值来判断声效段是否正在播放。调用执行接口的 IsPlaying 函数,并传递给它需要查询的段状态接口就可以实现。如果段正在播放,IsPlaying 返回 S_OK;反之,则返回 S_FALSE。

请注意在刚开始播放声效段时调用这个函数。有时 IsPlaying 函数太精确了,尽管它只是负责反映音频是否正在从声效段输出到扬声器。比如刚启动 PlaySegment 函数时,IsPlaying 函数可能因为延时而返回假值,它实际上表示已经开始播放音频,但还没有从扬声器里播放出声音。

3.3.8 释放音频

当完成对音频的操作后,需要释放音频,可以通过调用段接口的 Unload 函数来完成。它将从演奏对象中释放声效段的数据。

3.4 播放音频的其他工具

直到现在,我们一直把重点放在如何使用 DirectX API 来播放音频,但这并不是惟一途径,有时它也不是最好方法。本节介绍另外两种播放音频的方法。

3.4.1 DirectSound

替代 DirectMusic 的一种方法是使用低层的 DirectSound 接口。如果使用 DirectX,那么绝大多数时候是使用高层的 DirectMusic Loader 接口,一般没有必要使用低层接口。只有在不想使用 DirectX8 时,才使用 DirectSound(比如,希望游戏能够在 Windows NT 4.0 平台上运行)。下面介绍使用 DirectSound 接口时,如何加载和播放音频。

提示

有好几个 DirectX 示例程序演示了如何使用 DirectSound 接口来播放声音。在DirectX8 SDK 中,有两个文件(dsutil. cpp 和 dsutil. h)提供了许多工具,包括使用 DirectSound 缓冲区的 CSound 类的代码。

①初始化 DirectSound。如何初始化取决于所使用的 DirectSound 版本,一般的过程与上一章所介绍方法的类似。

②创建辅缓冲区并加载 WAV 文件。这一步的实现比较灵活,因为它涉及到解析 WAV 文件的格式。请考虑使用 DirectX 示例程序或者 Internet 提供的代码。

③调用创建的辅缓冲区的 Play 函数播放音频。

④最后,释放辅缓冲区。

Ch3p2_DirectSound 示例程序演示了如何使用 DirectSound 接口来播放音频。

3.4.2 Windows API Waveform Audio

如果不想使用 DirectX,还有另外一种播放 WAV 文件的选择:使用 Win32 API 的 Waveform Audio 函数。关于 Waveform Audio 的解释文本请查阅 MSDN,即在目录 SDK\Graphics 和 Multimedia Services\Multimedia Audio\Waveform Audio 下。

注意 Win32 API 的 PlaySound 函数,它可以从硬盘、资源或内存中播放音频。Ch3p2_WaveformAudio 示例程序演示了如何用这个函数从内存中播放音频。

> **提示**
>
> 如果想使用 PlaySound，你必须连接 winmm. lib 类库。请确保在项目设置的 linker 选项卡的类库列表中加入了“winmm. lib”，否则会得到连接错误。

3.4.3 Windows MCI

除了上述方法外，还可以使用 Win32 API 的媒体控制接口（MCI）来播放音频。MCI 被设计用来与各种多媒体设备交互，包括 WAV 音频播放器、MIDI 音序器、CD 音频播放器和数字视频播放器。

MCI 是一个功能很强的接口，它提供了多种关于播放音频的控制。特别是可以播放一个 WAV 文件的段，并且指定开始和结束的时间点。

在本书的后续部分，将介绍如何使用 MCI 来播放 CD。MSDN 中有更详细的内容，在目录 SDK\Graphics 和 Multimedia Services\Multimedia Audio\MCI 下。

3.5 本章小结

在本章中，学习了如何使用 3 个 DirectMusic 接口：加载器、声效段和演奏对象。也学习了创建引擎的基本规则，以及几种其他的播放声效的方法。

下一章将更深入地学习 DirectMusic，并且拓展音频引擎的功能，使之支持某些音频处理。

PDG

第4章

加载 WAV 文件



前三章已经向管理整个游戏声效的目标走了很长一段路。本章将继续深入介绍其他一些从驱动器上读取 WAV 文件并放入 DirectMusic 的方法。

4.1 剖析 WAV 格式

WAV 文件并不如想象中的那么简单，尽管 DirectMusic 屏蔽了其复杂性，但一旦离开微软的保护伞，就很难知道自己该怎么做。

因此，我们来浏览一下 WAV 文件的格式，请参考图 4.1 所示的文件字节示意图。想了解更多细节，可以查看光盘中提供的链接。

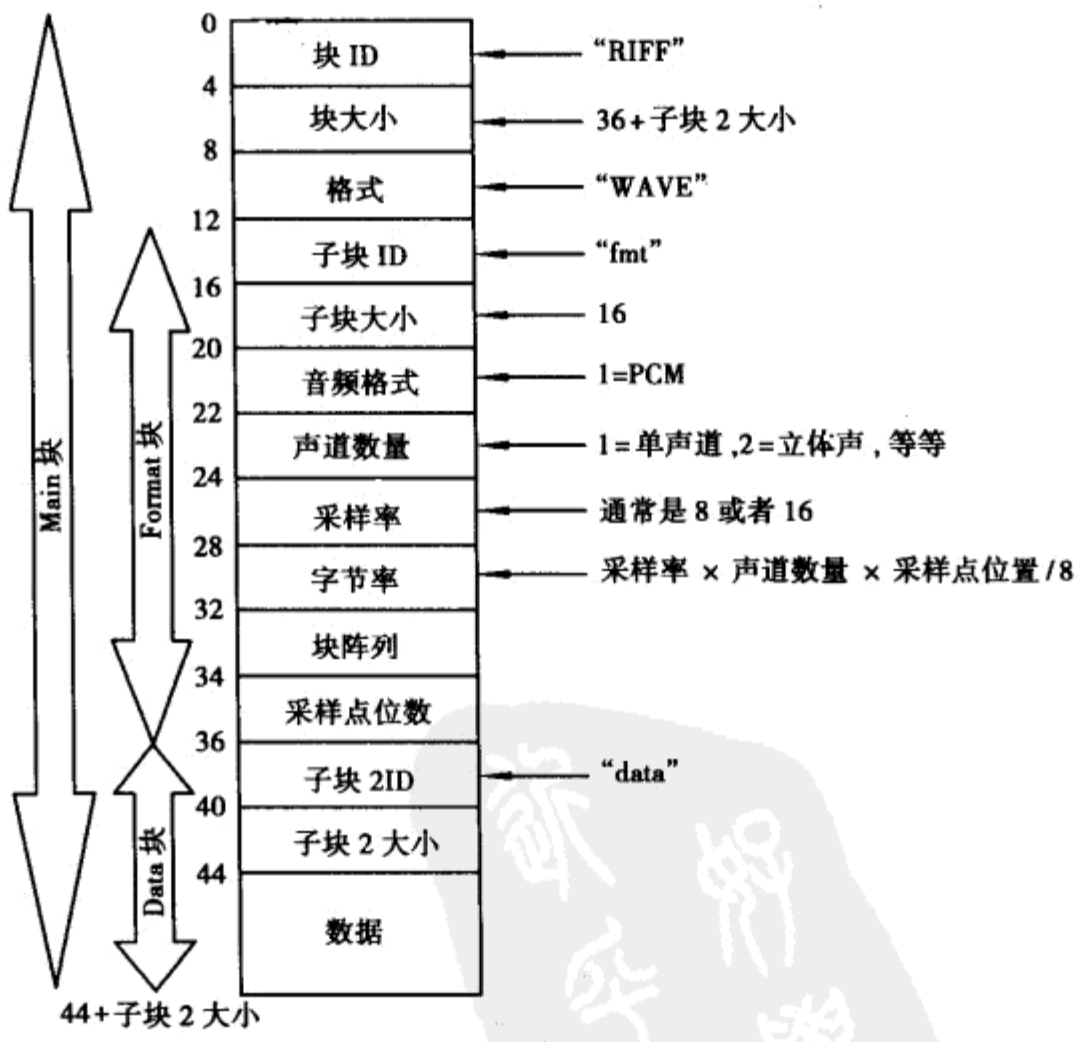


图 4.1 典型的非压缩 WAV 文件格式

4.1.1 资源交换文件格式 RIFFs

WAV 文件格式是微软的 RIFF 技术的一个子集,因此,要理解 WAV,必须先理解 RIFF。RIFF 表示资源交换文件格式(Resoure Interchange File Format),这种文件格式被用来保存多媒体数据。有趣的是,DirectMusic 音乐和其他音乐数据也采用 RIFF 格式来存储。

RIFF 文件格式比较复杂,但加载一个 WAV 文件时,并不需要完全理解它的格式。对于一般的 WAV 文件,只需要知道数据的每个元素是存放在什么地方的,这就是本节将要介绍的内容。如果对 RIFF 特别地感兴趣,可以搜寻 MSDN 中关于较高层次的 RIFF 体系结构,或查阅光盘中提供的链接。

4.1.2 块的概念

RIFF 文件的每个不同的部分被称作块。正像 Granny Smith 苹果有 4 017 种标签,每个块都有一个 4 字符的标识(块 ID),它告诉使用者这是哪一种块。这种特定的 ID 被称为 FOURCC(4 字符编码)。

一个 FOURCC 实际上是一个 32 位的整数(它的类型被定义为 DWORD),它封装了 4 个 8 位的 ASCII 值(如图 4.2)。

一个 WAV 文件由 3 个块组成。第一个是 main 块,它的 FOURCC 是“RIFF”。RIFF 块中的数据表明正在处理的是 WAV 格式的数据,而在 RIFF 块中有两个子块:“fmt”(注意在这儿有空格)和“data”。fmt 块包含了 WAV 文件的格式(8 位或者 16 位,立体声或单声道,采样率等)。Data 块包含了形成音频的实际字节数据。

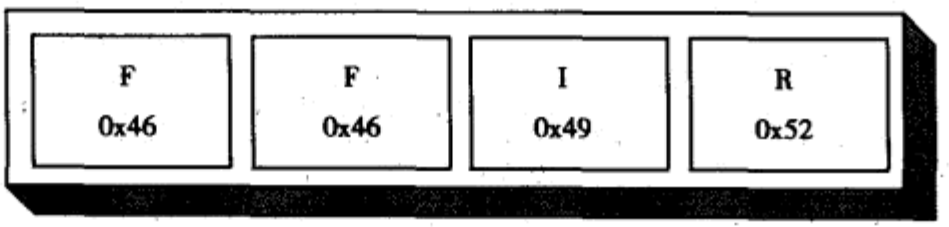


图 4.2 RIFF 的 FOURCC 字符编码(注意字母的顺序)

通常每个块有两部分。块的前 4 个字节是块的 FOURCC。因为 WAV 文件本身是 RIFF 文件,所以合法的 WAV 文件的前 4 个字节总是 RIFF。

提示

大写的 FOURCC 是标准格式;小写的 FOURCC 通常表示私有格式。

在每个块中,紧跟在块 ID 后面的 4 个字节表示块的长度。注意这个长度不包括块 ID 和表示长度的字节。换句话说,这个长度比整个块的实际长度小 8 个字节,因为前 4 个字节是块的 FOURCC,接下来的 4 个字节是块的长度。这一点非常重要,把长度值视为

“从这以后的数据的长度”,可以有助于记忆。

接下来的章节将描述 main 块、format 块和 data 块的细节。

4.1.3　Main 块

Main 块包含了 Format 块和 Data 块,它的 FOURCC 是 RIFF。

该块在段 ID 和段长度之后,只剩下 FOURCC,它表述了 RIFF 文件实际的内容。对于一个 WAV 文件,这 4 个字节总是“WAVE”。而对于其他格式的 RIFF 文件,将会是其他的代码。

Main 块实际上所描述的内容为:“我是一个 RIFF 文件,且包含一个 wave”。

请注意块的长度是另外两个块的长度加上包含格式数据的 4 个字节的和。

4.1.4　Format 块

现在,事情开始变得有趣了。Format 块包含了所有说明 Data 块的信息。如图 4.1 所示,该块表明音频格式、声道数量、采样率、字节率、块阵列和采样点位数。

(1)音频格式

音频格式占两个字节,其数据反映了 Data 块是否被压缩。对于非压缩的 WAV 而言,值为 1,这表示 Data 块的数据采用脉冲编码调制(PCM)进行存储。PCM 是 WAV 数据没有被压缩的另一种说法。

其他的值指出 WAV 文件是如何被压缩,见表 4.1。

表 4.1　常见的 Wave 格式 ID

WAV 格式	在 MMREG.H 中的定义	值
不确定(这种方法不好)	WAVE_FORMAT_UNKNOWN	0
脉冲编码调制	WAVE_FORMAT_PCM	1
自适应差分 PCM	WAVE_FORMAT_ADPCM	2
32 位浮点	WAVE_FORMAT_IEEE_FLOAT	3
CCITT G.711 A-law	WAVE_FORMAT_ALAW	6
CCT G.711 u-law	WAVE_FORMAT_MULAW	7
MPEG Layer 3(MP3)	WAVE_FORMAT_MPEGLAYER3	55

除了表中列出的格式以外,还有很多其他的格式,查看 MMREG.H 文件,可获取完整的列表。

正如所看到的,有很多可用格式,对它们一一进行编码不太可能。幸运的是,绝大多数的 WAV 文件都采用 PCM 格式,并且可以通过程序把一个非 PCM 格式的 WAV 文件转换成 PCM 格式。如果必须(或者希望)编写一个自己的 WAV 文件解析器,建议只考虑支持 PCM 格式。

提示

在网上有一些类库可以帮助你读取各种压缩的 WAV 文件格式。光盘中包含了一些这样的链接。

(2)声道数量

紧跟在音频格式后面的两个字节就是声道数量,它表示 WAV 文件有几个声道。如果是单声道的,就是 1 个声道;如果是立体声的,就是 2 个声道。也可能是其它数字,比如,一个 WAV 文件可以包含六声道的环绕立体声数据。但一般说来,最常见的还是单声道和立体声。

(3)采样率

采样率用 4 个字节表示,8 000 表示 8 000 Hz,44 100 表示 44.1 kHz,以此类推。

(4)字节率

字节率显示每秒钟的音频需要使用多少字节的空间。其值等于采样率乘以声道数,再乘以每个样本点的位数除以 8(每个样本点 16 位实际上就是每个样本点 2 字节)。

(5)块阵列

块阵列显示每个样本点的字节数。其值等于声道数乘以每个样本点的位数除以 8。

(6)采样点位数

如果采样点位数取值为 8,则表示处理 8 位的样本点,如果取值为 16 就表示处理 16 位的样本点,以此类推。8 和 16 是最常用的值。

4.1.5 Data 块

读完所有的格式数据,就确切地知道如何解释本块中的数据了。在 Data 块中,FOURCC 和块大小以后的部分就是原始的音频数据字节。如何解释这些字节依靠 Format 块。

对于非压缩的 WAV 文件而言,Data 块的长度总是等于样本点的数目乘以声道数,再乘

以每个样本点的位数除以8。

> **提示**
>
> 请记住,通常都不需要直接处理 WAV 文件格式,多数时候将使用 DirectMusic 加载器。

4.1.6 处理 WAV 文件

学习 WAV 文件的构成后,自然会想到增加一些新的对象类来增强音频引擎,那么请继续阅读下文。

(1) CWAVFile 类

通过 CWAVFile 类的方法可以直接管理 WAV 文件的数据:

```
class CWAVFile
{
public:
friend class CAudioManager;
void Init();
CWAVFile() { Init(); }
CWAVFile(const CWAVFile &r) { Init(); Copy(r); }
virtual ~CWAVFile();

int m_AudioFormat;
int m_NumberOfChannels;
int m_SampleRate;
int m_ByteRate;
int m_BlockAlign;
int m_BitsPerSample;
unsigned int m_DataLen;
CWAVFile &operator = (const CWAVFile &r) { Copy(r); return(*this); }
unsigned char * GetData() { return(m_Data); }
void SetData(const unsigned char * data, unsigned int len);
```

```
    void Load(const unsigned char * data);
    void Load(std::string filename);
    unsigned char * Save() const;
    bool Save(std::string filename) const;
    int GetTotalSize() const { return(44 + m_DataLen); }
    std::string GetInfo();
    protected:
    void Copy(const CWAVFile &r);  // Copy 方法的内容省略
    // data 块
    unsigned char * m_Data;
}
```

这个类保存了一个 WAV 文件的 Format 块和 Data 块。它以公共变量的形式提供了 Format 块中的各条信息，而用一个受保护的 m_Data 指针保存了 Data 块。这里没有为这个类创建访问器，因为这些数据都不会改变，但是如果更习惯于使用 Get/Set 方法，也可以加上。

提示

CWAVFile 类重载了一个操作符，这使得它可以拷贝 m_Data 指针。对一个动态分配内存的类重载 operator = 操作符，在C ++中是禁止的，这一点要小心。此外也要记住你的虚析构函数。

(2) 加载 CWAVFile 类

在理解了文件格式后，加载一个 WAV 文件到 CWAVFile 类就是按部就班的事了。注意 CWAVFile 类有两个加载方法：一个用于加载文件，一个用于加载内存中的数据。文件加载只是把文件读入内存中，然后调用内存数据的加载方法。它的实际工作代码如下：

```
void CWAVFile::Load(const unsigned char * data)
{
    FOURCC riff, wave, fmt, datacc;

    // 检查 ChunkID 是否是 RIFF
    memcpy(&riff, &data[0], sizeof(FOURCC));
    if (riff != mmioFOURCC('R', 'I', 'F', 'F'))
      { Throw("CWAVFile::Load: invalid WAV file data (missing RIFF header).");
```

```
// 检查 WAVE 格式
memcpy( &wave, &data[8], sizeof(FOURCC));
if (wave != mmioFOURCC('W', 'A', 'V', 'E'))
  { Throw("CWAVFile::Load: invalid WAV file data (not WAVE format).");}

// 检查 ChunkID 是否是 FMT
memcpy( &fmt, &data[12], sizeof(FOURCC));
if (fmt != mmioFOURCC('f', 'm', 't', ' '))
  {Throw("CWAVFile::Load: invalid WAV file data (FMT subchunk not found).");}

// 检查 ChunkID 是否是 DATA
memcpy( &datacc, &data[36], sizeof(FOURCC));
if (datacc != mmioFOURCC('d', 'a', 't', 'a'))
  {Throw("CWAVFile::Load:invalid WAV file data (DATA subchunk not found).");}

// 加载数据成员
memcpy( &m_AudioFormat, &data[20], sizeof(short int));
memcpy( &m_BitsPerSample, &data[34], sizeof(short int));
memcpy( &m_BlockAlign, &data[32], sizeof(short int));
memcpy( &m_ByteRate, &data[28], sizeof(int));
memcpy( &m_NumberOfChannels, &data[22], sizeof(short int));
memcpy( &m_SampleRate, &data[24], sizeof(short int));
unsigned int datasize = 0;
memcpy( &datasize, &data[40], sizeof(int));
SetData( &data[44], datasize);
}
```

这段代码是相当谨慎的。在做任何实际操作前,它都尽可能确保 Data 是指向一个合法的 WAV 文件。如果检查了所有的 FOURCC,它就使用 memcpy 从内存中将相关的数据取出来。最后,它通过调用 SetData 方法得到 WAV 文件数据。注意,这个类没有 Data 指针,它做的是一个拷贝操作。这意味着客户端应该尽可能早地释放这段内存空间。

mmioFOURCC 方法是一个方便的 Win32 函数,它是在 mmsystem.h 文件中定义的,它可以把 4 个字符依次放入 FOURCC 中。一个 FOURCC 实际上是一个 32 位的值,因此没有对它进行压缩,只是重新分配和构造某些灵活的字节。

(3)保存 CWAVFile 类

针对 CWAVFile 类进行操作的 Save 函数与 Load 函数是对称的。有两个 Save 函数,一个用于保存到磁盘上,另一个用于保存到内存中。保存到磁盘上的函数先调用保存到内存中的函数,然后再保存内存中的数据。

这里,不再讲述 Save 函数的代码,需要说明的是,Save 函数的任务是分配足够的内存空间、重新创建 FOURCC、将 CWAVFile 类中的变量放到相应的位置并存储WAV数据。当这些事情做完以后,Save 函数就返回一个所创建内存的指针。当 Save 返回指针后,调用者就拥有这块内存空间。

提示

为了获得额外的好处,可以尝试将一个读取压缩格式 wave 文件的类库集成到音频引擎中。如果你能让音频引擎从磁盘读取一个 wave 文件;并且在 CWAVFile 对象加载该文件时自动解压缩 wave 文件,这将是一件很酷的事情。

Ch4p1_WAVFileXRay 示例程序演示了如何将一个 WAV 文件放入内存中。

4.2 加载 WAV 的其他方法

现在,对 WAV 文件的格式有了基本的了解,再回过头来看 DirectMusic 模式,学习一些不同的加载 WAV 文件的方法。上一章中,学习了如何从磁盘加载 WAV 文件,现在学习如何从内存或者应用程序资源中加载它们。

4.2.1 从内存加载

使用 DirectMusic 的 Loader 对象从内存而不是从磁盘读取数据。这是一种很强的功能,因为有了这个功能,只需要将整个文件读入内存就可以随时保存 WAV 文件。可以将 WAV 文件保存成 ZIP 文件,或者是一种自定义的文件格式,或者可以从 Internet 下载。

如果将一个 WAV 文件读入内存,可以依靠 DirectMusic 的 Loader 方法来解析 WAV 文件并得到一个声效段。

在第3章“WAV 音频的播放”中,使用 IDirectMusicLoader8 的 LoadObjectFromFile 函数从磁盘加载一个 WAV 文件。而 LoadObjectFromFile 函数只是一种简写的形式,实际上是依靠加载器的 GetObject 函数,该函数可以从磁盘或者内存中读取数据。

为了使用 GetObject 函数,必须首先构建一个 DMUS_OBJECTDESC 结构,这个结构告诉

DirectMusic 需要操作的对象在哪里并且如何加载它。从内存中加载一个对象,需要在 DMUS_OBJECTDESC 结构中设置两个重要变量。第一个是把 pbMemDate 字节指针设置成你的“文件”在内存中的位置。第二个是设置 llMemLength 变量,它告诉 DirectMusic 你的内存块的长度。

应用这些知识,可以创建另一个 LoadSound 函数来增强音频管理器,其代码如下:

```
CSoundPtr CAudioManager::LoadSound(unsigned char * data, int datalen)
{
    HRESULT hr;
    CSound * snd = new CSound(this);
    DMUS_OBJECTDESC desc;
    memset(&desc, 0, sizeof(DMUS_OBJECTDESC));
    desc.dwSize = sizeof(DMUS_OBJECTDESC);
    desc.dwValidData = DMUS_OBJ_MEMORY | DMUS_OBJ_CLASS;
    desc.guidClass = CLSID_DirectMusicSegment;
    desc.llMemLength = datalen;
    desc.pbMemData = data;
    hr = m_Loader -> GetObject(&desc, IID_IDirectMusicSegment8,
      (void * * )&snd -> m_Segment);
    ThrowIfFailed(hr, " CAudioManager::LoadSound(unsigned char * data): GetObject
    failed. ");
    snd -> m_OriginalData = data;
    return(CSoundPtr(snd));
}
```

这段代码的开始部分与原来的 LoadSound 函数类似,它创建了一个新的 CSound 对象。随后,代码的执行发生了变化:原来的 LoadSound 调用 LoadObjectFromFile,而这段代码则构建 DMUS_OBJECTDESC 结构并调用 GetObject。

并非 DMUS_OBJECTDESC 结构的每一个变量都需要设置。只有 dwSize 变量和 dwValidData 标记是必需的,其他变量是否需要设置则依赖于 dwValidData 标记。把 dwValidData 标记指定为 DMUS_OBJ_MEMORY,告诉 DirectX Audio 从内存中加载对象。指定这个标记则必须分别设置 llMemLength 和 pbMemData 的值为数据大小和数据本身。

将 dwValidData 标记指定为 DMUS_OBJ_CLASS,则告诉 DirectX Audio 加载 guidClass 变量所指定类型的对象。代码中设置 guidClass 变量为 CLSID_DirectMusicSegment,则告诉 DirectX Audio 应该返回一个段接口。

在调用 GetObject 后,代码设置新音频对象的 m_OriginalData 变量。它以管理器的身份告诉音频:“你的段对象正在使用程序分配的内存,因此,在你被删除时要释放这段内存空间”。不能在调用 GetObject 后立即释放内存,因为 DirectXAudio 是在段里面使用这段内存。它不是拷贝这段内存,而是在段的生命周期中将数据放在那里。因此,当段使用完毕后,需要释放内存。

Ch4p1_WAVFileXRay 示例程序演示了这个方法的使用。

4.2.2 从 CWAVFile 加载

目前,音频引擎有 CWAVFile 对象类和 Csound 对象类,可以将一个 CWAVFile 对象转换成 CSound 对象。如果这样做,那么在运行游戏时就可以用 CWAVFile 来创建声效,然后在游戏中把这些声效放到 CSound 中播放。

这种集成实际上是很简单的,只是让 CWAVFile 保存自己到内存中,然后把这段内存交给 DirectMusic 的 Loader,在内存中创建一个声效段。下面是代码:

```
CSoundPtr CAudioManager::LoadSound(const CWAVFile &wavfile)
{
  unsigned char * savedata = wavfile.Save();
  return(LoadSound(savedata, wavfile.GetTotalSize()));
}
```

它只有两行代码,很简便吧。只是要记住不要释放 Save 函数分配的内存就行了,因为 CSound 要使用这段内存。实际上,因为 LoadSound 函数的这段内存被设置成了 CSound 的 m_OriginalData变量,所以没有什么可以担心的。当 CSound 被删除时,将释放这段内存。

4.2.3 从资源中加载

从资源中加载一个 WAV 文件和从内存中加载如出一辙。只需调用几个 Win32 API 函数来得到指向自定义资源的指针(如 WAV 文件),然后将指针通过前面描述的 DMUS_OBJECTDESC 结构交给加载器。下面是代码的片断:

```
CSoundPtr CAudioManager::LoadSound(HMODULE hmod, char * type, WORD resID)
{
    CSound * snd = new CSound(this);
    HRESULT hr;
    DMUS_OBJECTDESC ObjDesc;

    HRSRC hFound = FindResource(hmod, MAKEINTRESOURCE(resID), type);
```

```
if (NULL == hFound) {
  Throw("CAudioManager::LoadSoundFromResource: couldn't find resource!");
}
HGLOBAL hRes = LoadResource(hmod, hFound);

ObjDesc.dwSize = sizeof(DMUS_OBJECTDESC);
ObjDesc.guidClass = CLSID_DirectMusicSegment;
ObjDesc.dwValidData = DMUS_OBJ_CLASS | DMUS_OBJ_MEMORY;
ObjDesc.pbMemData = (BYTE *) LockResource(hRes);
ObjDesc.llMemLength = SizeofResource(hmod, hFound);

hr = m_Loader -> GetObject(&ObjDesc, IID_IDirectMusicSegment8,
  (void **) &snd -> m_Segment);
ThrowIfFailed(hr, "CAudioManager::LoadSoundFromResource(HMODULE hmod,
  WORD resID): GetObject failed.");
return(CSoundPtr(snd));
}
```



图 4.3 工作区的一个 ResourceView 选项

这段代码跟其他 LoadSound 函数很相似。主要区别是在构建 DMUS_OBJECTDESC 结构以前,代码必须找到并加载资源。调用 FindResource 返回一个 HRSRC 句柄,这个句柄将在后面的 LoadResource 中使用。LoadResource 返回 HGLOBAL,这将被 LockResource 使用。LockResource 得到 HGLOBAL 并返回指向内存中资源数据的指针。代码将这个指针放入 DMUS_OBJECTDESC 结构的 pbMemData 变量中,然后继续执行。注意,这个操作是通过调用 SizeofResource 来判断一个资源的大小。

Ch4p2_WAVInResource 示例程序演示了这个方法的使用。

如何把 WAV 文件放到资源中呢?下面给出添加一个 WAV 文件到程序资源的方法。

如果没有一个资源文件,那么需要先创建一个并把它加到项目中。在 DevStudio 里,从 File 菜单选择 New,然后选择文件类型为 Resource

Script。惯用做法是使程序主资源脚本文件的名称与项目的EXE、DSP和DSW文件名一样。比如,Ch4p2_WAVInResource资源脚本文件的名称就是Ch4p2_WAVInResource.rc。关闭DevStudio所打开的窗口。

当创建了资源文件以后,可以在工作区看到一个ResouceView选项,如图4.3所示。点击这个选项,然后在最上端的目录上点击右键,如图4.4所示。从文件浏览器中选择想加入的WAV文件。

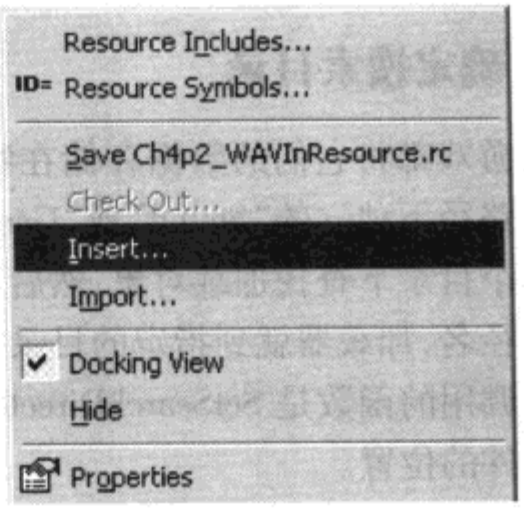


图4.4 把一个WAV文件导入到应用程序资源中

下一步需要告诉DevStudio资源的种类,可以选择所需要的资源类型,但与此同时应该记住传递给音频管理器的LoadSound函数相同的类型。这里用"WAVE"来表示WAV文件。

如果一切顺利,可以看到自定义资源已罗列在一个目录中,该目录就是上一步所指定的名字。

提示

DevStudio默认不会在RC文件中保存整个自定义资源,而只保存一个编号,这个编号指示了原始文件的位置。通常,这种做法是好的,但如果需要让DevStudio在资源文件中直接保存所选择的WAV文件拷贝,那么可以右键单击所需的资源,选择Properties,然后将看到如图4.5所示的对话框。

取消选择External File复选框,则资源文件将存储在脚本文件中。注意将资源存储在RC文件中会占用更多的空间——大概是正常大小的5倍,因为它们是"ASCII编码的"(比如,一个单一的16进制字节30转变成ASCII字符串是"0x30")。

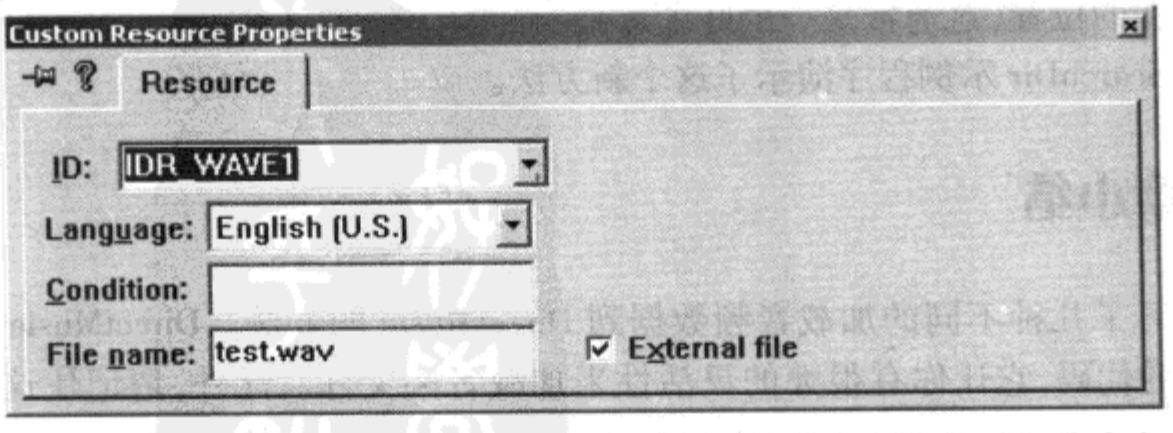


图4.5 取消选择External file复选框,使WAV直接嵌入到程序资源脚本中

编译的程序和资源将被包含在 EXE 文件中。

确保在每个需要使用资源的 CPP 文件中包含 resource. h。resource. h 使用#define 把在资源管理器中指定的资源名称绑定到资源 ID(这是 Win32 API 所需要的)。

4.2.4 确定搜索目录

多数游戏都将它们的数据存储在特定的目录下。很多专业游戏将它们的音频数据存储在安装路径下独立的“Sound”或“GameAudio”目录下。有个技巧可简化这件事:先告诉加载器在哪个目录下查找加载对象,然后,只需要提供每个要加载的音频文件名,而不需提供完整的路径名,加载器就到指定的目录中查找文件。

需要调用的函数是 SetSearchDirectory。下面给出一个新的音频引擎方法,它告诉加载器音频文件的位置。

```
void CAudioManager::SetSearchDirectory(std::string dirname)
{
    // 将路径名转换成 wide-string
    WCHAR widedirname[MAX_PATH];
    DXUtil_ConvertGenericStringToWideCch(widedirname, dirname.c_str(), MAX_PATH);
    HRESULT hr = m_Loader -> SetSearchDirectory (CLSID_DirectMusicSegment, widedi-
    rname, true);
    ThrowIfFailed ( hr, " CAudioManager:: SetSearchDirectory: SetSearchDirectory ( )
    failed. " );
}
```

这段代码并不长,只是将路径名转换成宽字符串,然后调用 SetSearchDirectory 函数。该函数的第一个参数告诉 DirectX Audio 准备设置的路径包含什么类型的对象(此处是段)。第二个参数是目录的名称。最后一个参数如果是真值,则告诉 DirectX Audio“忘记”它上次加载对象所在的位置(这通常是一个好主意)。

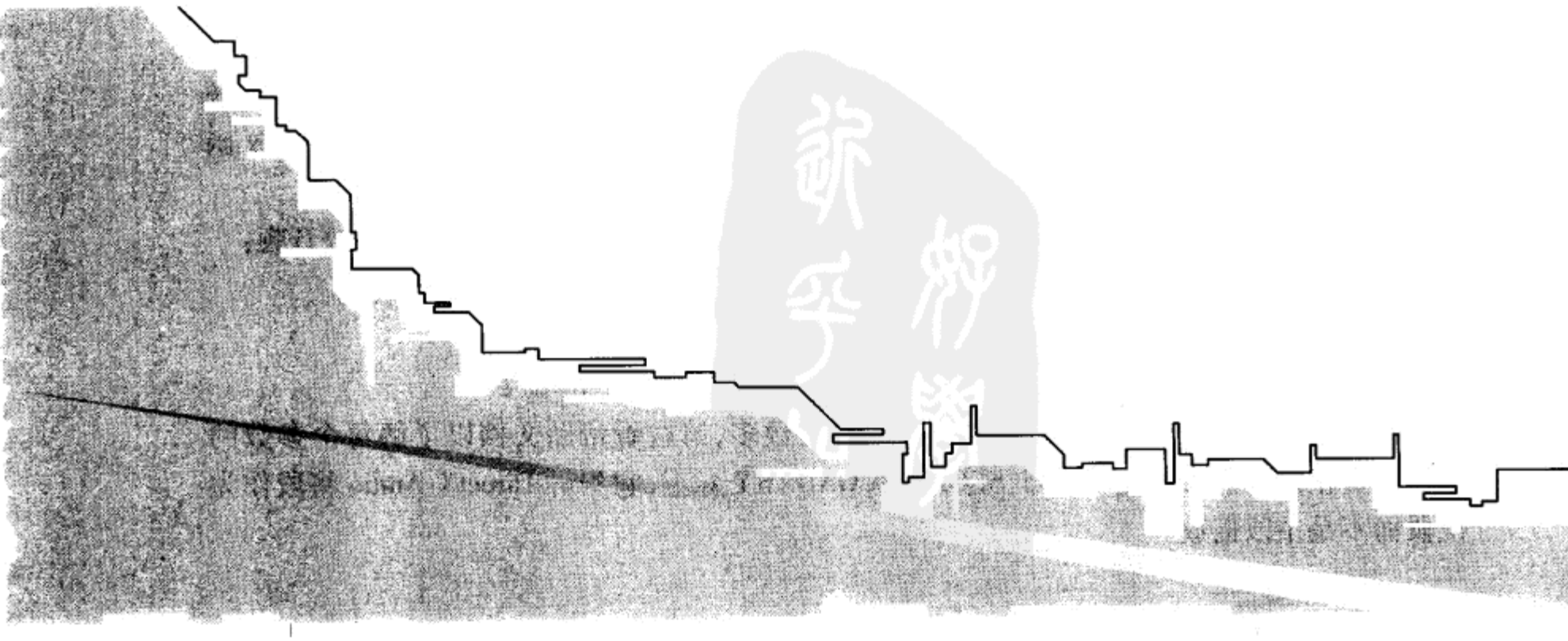
Ch4p3_SearchDir 示例程序演示了这个新方法。

4.3 本章小结

本章学习了几种不同的加载音频数据到 DirectMusic 的方法。DirectMusic 加载器是一段功能强大的代码,它让你有很大的灵活性来加载音频文件。此外,请记住这里所有的加载器技术不仅可以用于声效,也可以用于音乐。

第5章

声效控制技巧



最终完成音频引擎的声效部分的设计还缺少几个重要功能。这些功能涉及到声效控制：调节音量、连续播放音频等。

这一章将讲授让声效变好的小技巧。

5.1 连续播放音频

读者可能已经注意到：到目前为止，书中设计的音频引擎还不能重复播放相同的音频。换句话说，如果只有一段铃声，现在还不能连续响铃 3 次，还只能让铃声开始响，然后结束，如此往复 3 遍。本节将讨论如何解决这个问题。

5.1.1 DirectMusic 的主段与次段

DirectMusic 有主段与次段的概念。它们的区别是次段可以有多个，而主段只能有一个。

主段被认为是游戏的“主音轨”，而声效或其他声音则通常使用次段，然后通过主段来播放。主段通常指出了音乐的参数，比如音乐节拍、旋律等。接下来将学习这方面的知识，现在，只需要记住多次播放声效的惟一途径是让声效作为次段播放。

5.1.2 用 PlaySegmentEx 替代 PlaySegment

上一章，使用 DirectMusic 演奏对象的 PlaySegment 来播放音频，这种方式只能播放音频一遍，不能采用这种方法进行多次播放，但可以使用 PlaySegmentEx 函数。PlaySegmentEx 函数可以告诉 DirectX Audio 将一个段作为主段播放还是次段播放。

在 CSound 的 Play 函数中使用 PlaySegmentEx（替代 PlaySegment）可以多次播放音频：

```
m_Manager -> GetPerformance( ) -> PlaySegmentEx( m_Segment, NULL, NULL,
DMUS_SEGF_SECONDARY, 0,
(IDirectMusicSegmentState ** )&m_SegmentState, NULL, NULL);
```

PlaySegmentEx 函数的参数比 PlaySegment 函数多，请查看帮助文档以了解每个参数的含义。真正重要的参数是 DMUS_SEGF_SECONDARY 标记，它告诉 DirectX Audio 将段作为次段而不是主段播放。

5.2 音频实例程序

到现在为止,还没有完成连续播放音频的任务。上节介绍了如何使 DirectMusic 多次播放一段音频,本节介绍如何使音频引擎适应这个新要求,即可以不止一遍地播放音频。

音频引擎现在遇到的问题是它只保存了一个段状态(IdirectMusicSegmentState8)。一旦播放音频,m_SegmentState 就通过一个接口被赋值为新的段状态;在播放其他音频时,也会发生相同的事情,并且永远丢失前一个段状态。更糟糕的是,前一个音频的段状态并没有被释放,这是一个资源漏洞。

解决这些问题,音频引擎需要一个新类 CSoundInstance。这个类代表当前正在播放的音频。每次音频播放时,Csound::Play 都将创建(并返回)一个新的 CSoundInstance。这样调用 Play 的代码就可以保存特定的音频实例。

这个类在循环播放声效时很有用(循环播放在下一节介绍)。通常调用 Play 播放非循环声效的很多代码在播放时并不关心音频。就好像当一个玩家在射击时,射击代码将调节音频参数(音量)并让它播放。

对于循环播放音频,其模式有些不一样。循环播放音频通常需要响应游戏事件。例如在一个实时战略游戏中的能量产生器发出嗡嗡声的情形:刚开始的时候,嗡嗡声几乎听不到,而当玩家建造了一个超级武器工厂时,能源需求量升高,于是嗡嗡声也要相应变大了。

在继续学习本书时,音频实例会变得很重要。现在,不会涉及太多的音频实例,但开始学习 3D 音频时,它就很重要了。每个音频实例将包含一个 3D 的位置,并且可以调节其他的一些属性。

接下来的一节将看到 Ch5p2_SoundInstances 示例程序。

> **提示**
>
> CSound 和 CSoundInstance 之间的关系是游戏编程中的一个常见模式。在 2D 游戏中,有在游戏世界中跑动的小精灵和表示小精灵外观的位图。这使得你拥有多个使用相同位图的小精灵,并且不用复制位图数据。
>
> 在本书的音频引擎里,CSound 就如同位图,而 CSoundInstance 就如同一个小精灵。

5.2.1 创建 CsoundInstance 类

CSoundInstance 是一个简单的对象类,因此在这里不详细解释它。在开发环境中,CSoundInstance 有 Init 函数和 UnInit 函数、一个受保护的段状态变量、一个标准的音频管理

器指针(用于 IsPlaying)以及一个指示对象是否合法的标记。

跟 CAudioManager 类一样,CSoundInstance 类也采用两阶段创建。这使得游戏可以预分配将在以后使用的音频实例。比如说在一个 CTank 类中,可以有一个统计已分配数量的变量 m_EngineSound:

```
class CTank
{
    public:
    CTank( ) { }   // 不做任何操作
    virtual ~CTank( ) { }   // 不做任何操作
    protected:
    CSoundInstance m_EngineSound;
}
```

如果 CSoundInstance 是象 CSound 那样设计的(一阶段创建),那么它将不能实现应有的功能,因为 CSoundInstance 需要指定音频的来源作为构造函数的参数。所以 Init 函数和 UnInit 函数,包括 m_Valid 标记都需要用两阶段创建。此外,还剩下 IsPlaying 函数、m_SegmentState 变量和相关的函数 GetSegmentState。IsPlaying 函数与 CSound 很相似,其代码如下:

```
bool CSoundInstance::IsPlaying( )
{
    assert(IsValid( )); if (! IsValid( )) { return(false); }
    return(GetAudioManager( ) -> GetPerformance( ) -> IsPlaying(
  NULL,m_SegmentState) == S_OK);
}
```

演奏对象的 IsPlaying 函数有两种方式,它既可以检查一个段是否正在播放,又可以检查一个段的状态。为了区别这两种方式,将段作为第一个参数,而把段状态作为第二个参数。因为 CSoundInstance 有段状态,所以它传递的第一个参数为 NULL,即什么都没有。

提示

在 IsPlaying 方法中,注意 IsValid 检查。使用两阶段创建的一个缺点是在每个方法中,在准备进行操作以前,必须首先确保得到已完全初始化的对象。这样,如果某人在实例初始化以前调用了 IsPlaying 函数,就不会引起程序冲突。

5.2.2 调整 Csound 类

现在 CSoundInstance 已经编好了,需要对 CSound 做一些变动。CSound 不再需要 m_SegmentState 变量,因为它已经被 CSoundInstance 处理。此外,CSound::IsPlaying()需要改变

成使用段而不是使用段状态。下面是新的代码：

```
bool CSound::IsPlaying( )
{
    if (NULL == m_Segment) return(false);
    if (NULL == m_Manager) return(false);
    return(m_Manager -> GetPerformance( ) -> IsPlaying(m_Segment, NULL) == S_OK);
}
```

在调用 IsPlaying 时进行调整，将段作为第一个参数，而 NULL 作为第二个参数。这告诉 DirectX Audio 如果有音频实例正在播放就返回 S_OK。

最大的改动还是 Play 函数，下面是它的新代码：

```
bool CSound::Play(CSoundInstance * newinst)
{
    if (NULL == m_Segment || NULL == m_Manager) {
      if (newinst) newinst -> UnInit( );
      return(false);
    }

    static CSoundInstance dummy;
    if (NULL == newinst) newinst = &dummy;
    m_Segment -> Download(m_Manager -> GetPerformance( ));

    // 初始化音频实例，这也将设置音频播放
    newinst -> Init(this);
    return(true);
}
```

如果是初学者，请注意参数 newinst，它是一个指向保存待播放音频实例的指针。在 CSound 类的声明中，newinst 参数的默认值是 NULL，因此代码不用考虑保持一个音频实例，只需要调用 Play() 就可以一直做下去。

如果 newinst 是 NULL，那么代码使用一个虚拟的静态 CSoundInstance 来初始化音频并播放它。否则，代码使用传递给它的 newinst。

请注意 CSoundInstance::Init() 需要一个 CSound 指针。Init 函数现在还没有使用 CSound 指针做任何事情，但在后面的章节中会做。

Ch4p4_ConcurrentPlayback 实例程序演示了这些新代码如何协同工作。

5.3 循环播放音频

现在,循环播放音频已经是很容易了。在 DirectMusic 以前,循环播放音频的通常做法是开始播放音频,然后经常检查音频是否仍在播放。如果音频没有播放了,就重新开始播放。

这种做法最大的问题是为了使循环播放没有延时,必须以很短的时间间隔来检查音频,近似于每秒 10 次。对于一个音频可能没有多大影响,但是对于好几个音频而言,这会消耗掉很多 CPU 资源。

一种更简单的做法是调用段接口的 SetRepeats 函数,可以把标记 DMUS_SEG_REPEAT_INFINITE 传给函数 SetRepeats,告诉 DirectMusic 无限循环地播放一个段。或者可以传进一个整数,该整数指定了循环播放的次数。

如果需要更好地控制循环,可以使用段的 SetLoopPoints 函数,指定循环播放开始和结束的位置。但是这可能不会像你期望的那样执行,见图 5.1。即使是使用 SetLoopPoints 指定了开始和结束的位置,整个段依旧会被播放。DirectMusic 将会从段的开头开始播放,一直到所指定的循环结束点,然后返回到所指定的循环开始点进行循环播放,循环的次数由 SetRepeats 设置,最后播放从循环结束点到段结尾的部分。换句话说,SetLoopPoints 只适用于循环播放;段仍旧会播放它在循环里面和外面的部分。

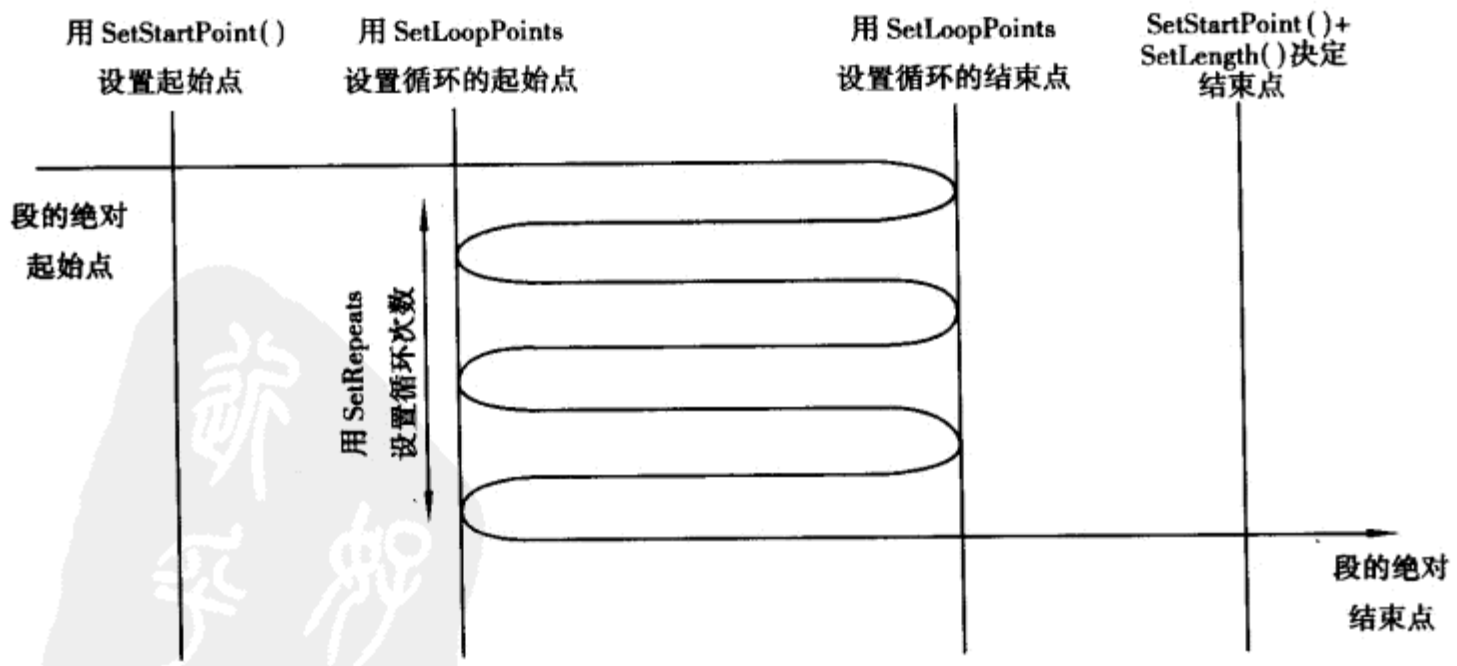


图 5.1 DirectMusic 使用几种不同的参数来控制段的循环播放

如果想控制实际的开始点和结束点，则需要使用段接口的 SetStartPoint 和 SetLength。默认情况下，加载器设置开始点是段的开头，而长度是整个段的长度。

Ch5p3_Looping 实例程序演示了如何循环播放一个音频。

提示

对于设置循环播放参数的方法，有与之对应的 Get 方法。

5.4 用 DirectX Audio 控制音量

游戏经常会根据游戏事件改变音频的音量。本节讲解如何使用 DirectX Audio 来改变音量。

5.4.1 音频通道

在 DirectMusic 中，音量被捆绑到被称为音频通道的对象上。一个音频通道包含所有影响段播放的信息，但不是段的必要组成部分。一个经典的例子就是音量。

提示

一个3D 音频的位置也是作为音频通道对象的一部分存储。关于这些将在后面学习。

音量控制没有必要驻留在段中。否则，当你想用不同的音量播放同一个段时，不得不复制段对象，这将造成一种可怕的资源浪费。因此，作为替代的方法，DirectMusic 将大部分音频实例的控制放到了称为音频通道的对象中。

当前音频引擎已经为它所播放的音频使用“默认音频通道”。默认音频通道是在调用 InitAudio 时创建的。这种方法可以用在简单的音频播放上，但是如果想改变一个特定音频的音量，就将会遇到麻烦：所有音频都使用同一个音频通道，那么当某个音频改变音频通道的音量时，它就间接地改变了其他所有音频的音量（参见图 5.2）。

解决这个问题的方法有两种，它们都涉及到使用多个音频通道。其方法一是创建一个音频通道“池”，当播放一个音频时，程序代码选择一个空闲的音频通道并将音频分配给它。对于初学者而言，这种方法让程序员比较容易查看和管理所有的音频通道，而不需要遍历所有正在播放的音频。这也是这个方法受到欢迎的原因。

另外一种可选方法是本书在音频引擎中所实现的方法：每当播放一个音频时创建一个新的音频通道。当音频播放完以后，音频通道还继续存在，直到它被释放为止。因为音频通道是相对轻量级的类，这种方法是可行的，它使程序员在游戏逻辑里可以只处理很少的分配操作。这种方法比前一种更易于编码和学习。

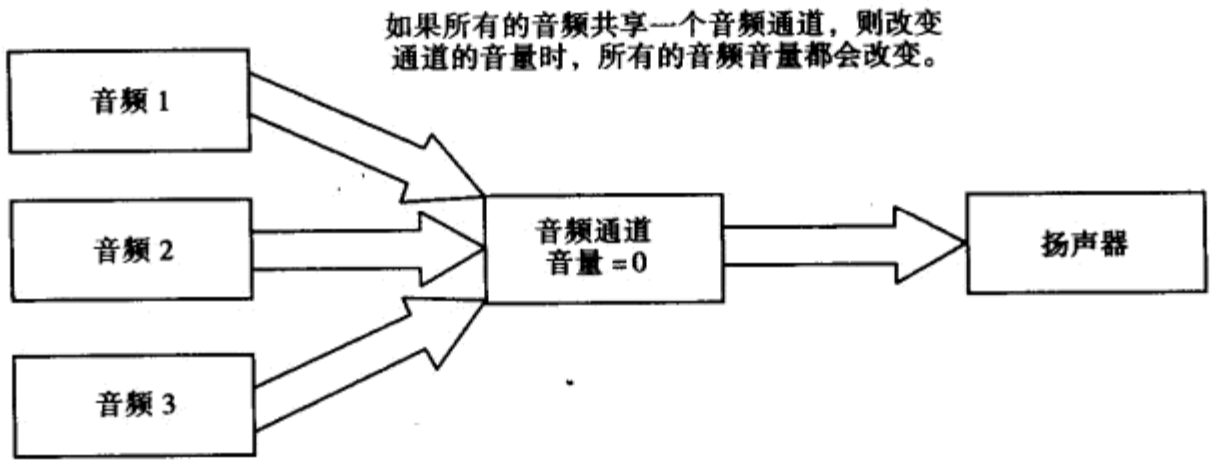


图 5.2　音频既可以共享音频通道也可以共享音量控制

为了适合音频通道,需要对 CSound 的 Play 函数做一些改动。下面是修改后的代码:

```
bool CSound::Play(CSoundInstance * newinst)
{
    HRESULT hr;
    if (NULL == m_Segment || NULL == m_Manager) {
        if (newinst) newinst -> UnInit();
        return(false);
}

static CSoundInstance dummy;
if (NULL == newinst) newinst = &dummy;

IDirectMusicPerformance8 * perf = m_Manager -> GetPerformance();
IDirectMusicAudioPath8 * pPath = NULL;
```

```
IUnknown * pConfig;

  if (SUCCEEDED(m_Segment -> GetAudioPathConfig(&pConfig)))
  {
     // 这里有一个内嵌的音频通道配置
     hr = perf -> CreateAudioPath(pConfig, TRUE, &pPath);
     pConfig -> Release();
     ThrowIfFailed(hr,"CSound::Play:Couldn't create audiopath using embedded config.");
  }
  else {
    // 没有配置……就创建我们自己的!
    hr = perf -> CreateStandardAudioPath(DMUS_APATH_DYNAMIC_STEREO,
       CAudioManager::NUMCHANNELS, TRUE, &pPath);
    ThrowIfFailed(hr, "CSound::Play: Couldn't create standard audiopath.");
  }

  m_Segment -> Download(perf);

  // 初始化音频实例(设置音量,等等),这也将使音频进行播放
  newinst -> Init(this, pPath);
  return(t rue);
}
```

代码的开始部分没有进行修改,但是当它得到演奏对象并放入 perf 中后,处理开始发生改变。首先,它查看段中是否有音频通道的配置。config 对象就像是音频通道的 DNA,因此如果代码找到了配置,它为创建音频通道所要做的就是将配置传给 CreateAudioPath,而 CreateAudioPath 将新的音频通道放入 pPath 变量中。WAV 文件本身没有音频通道配置,但在 DirectMusic Producer 中创建的音频段通常都有。

如果在段中没有配置,代码就创建一个新的标准音频通道,并且把这个接口放在 pPath 中。因此无论哪种方式,到代码加载段时,它已经有一个合法的、特定音频实例的、存放在 pPath 中的音频通道。在代码的最后一行,代码把 pPath 变量传给音频接口的 Init。下面是 Init 修改后的代码:

```
void CSoundInstance::Init(CSound * sound, IDirectMusicAudioPath8 * path)
{
```

```
    HRESULT hr;
    m_AudioPath = path;
    m_Valid = true;
    // 根据音频的音量设置音量
    SetVolume(sound -> GetVolume());

    // 现在所有的事情都设置好了,继续往下走并且播放音频
    hr = GetAudioManager() -> GetPerformance() -> PlaySegmentEx(
      sound -> GetSegment(), NULL, NULL,
      DMUS_SEGF_SECONDARY, 0,
      (IDirectMusicSegmentState **)&m_SegmentState,
      NULL, path);
    ThrowIfFailed(hr, "CSoundInstance::Init: PlaySegmentEx failed.");
}
```

Init 函数将音频通道接口放在 m_AudioPath 变量中。然后将此接口作为最后一个参数传递给 PlaySegmentEx。这将告诉 DirectMusic 使用该音频通道来播放音频。

当设置音频的音量时,Init 函数在 SetVolume 中使用了 m_AudioPath 接口,接下来将学习这个方法。

5.4.2 段的特定音量

前面提到通过音频通道来控制一个音频的音量。现在就讲解如何得到并且设置一个音频通道的音量。

(1)设置音量

IDirectMusicAudioPaht8 接口有一个函数叫做 SetVolume。这个函数有两个参数:一是音量,准确地说是分贝衰减,用几百分贝表示;二是音量发生变化的持续时间,这使得音量衰减很容易。CSoundInstance 的 SetVolume 函数使用音频通道的 SetVolume 函数来改变播放音频的音量。

这个设计中,在音频引擎里创建了一些更高级的功能。如果查看 CSound 类的声明,就会发现它也有一个 SetVolume,这使得可以对特定音频的新实例指定默认的音量。CSound 并没有做太多的操作,它只是记住所指定的值。实际的操作是在 CSoundInstance::Init 中完成的。请注意这行代码:

```
SetVolume(sound -> GetVolume());
```

这行代码设置音频通道的音量为当前 CSound 的音量大小。

提示

请注意在 CSoundInstance::Init 中操作的次序。特别要注意它在播放音频前设置了音频通道的音量。这样就确保能用正确的音量播放音频。当代码调用 PlaySegmentEx 设置完音量,可能稍微等一会儿,用户就可以听到最高音量的音频。操作的次序是很重要的——这一点切记切记。

(2)查询音量

查询音频通道的音量十分简单,只需调用音频通道接口的 GetVolume 函数。GetVolume 有一个参数,它指向一个长整数的指针,这个长整数被赋值为音频通道当前的音量值。

(3)其他音量

音频通道音量当然不是可以调节的惟一音量。在 DirectSound 缓冲区中也有音量,可以通过 IDirectSoundBuffer8 的 SetVolume 和 GetVolume 进行访问。

为什么会有这样的音量范围?

很多程序员都习惯用 0 ~ 1、或 0 到某个正数作为范围,但为什么 DirectMusic 的音量控制范围是 -9 600 ~ 0?

首先,范围是用分贝的百分之一来表示的。也就是说,-100 是 -1 分贝,而 -9 600 是 -96分贝,这样可以更准确。那为什么会是负数呢? 在过去模拟记录的时代,音频工程师用负分贝数来衡量设备的质量。值为 0 表示完美无缺,但不可能,因为它意味着设备在记录音频的操作中没有损失掉任何东西。随着值从 0 开始下降,设备的性能也下降。

正常的人耳可以听到大概 120 分贝范围内的声音。这意味着如果把扬声器的音量调大到足以永久损伤听力的程度,将导致如果以 0 音量播放音频,那么只能听到以 -120 音贝播放的声音。当然一个更敏感的范围比这个稍小一些,是 -96 ~ 0 分贝。

减小音频音量惟一的方法被称为衰减。实际上如今所有的音频硬件都是使用衰减来实现音频控制的。换句话说,声源的音量不会变得越来越大,只会变得越来越小。

只控制一个音量而不用去管其他音量的方法是最简单的。如果既要调节 DirectSound 缓冲区的音量,又要调节声效段的音量,则实现方法变得比较复杂。因此,通常只设置声效段的音量。

5.4.3 全局音量

在 DirectMusic 中还有其他可以控制的音量。DirectMusic 允许指定演奏对象的音量作为全局音量。这当然不会像调用 IDirectMusicPerformance8 的 SetVolume 函数那样简单。

可以通过调用演奏对象的 SetGlobalParam 来设置音量。把 GUID_PerfMasterVolume 作为第一个传递参数,把所希望的音量作为第二个传递参数,长度作为第三个传递参数。

演奏对象所使用的音量范围与音频通道的音量范围不同。具体范围取决于软件合成器。对于微软的内置合成器,音量用分贝表示,范围从 -200 ~ +20 分贝。这意味着实际范围是 -20 000 ~2 000,而不是音频通道的 -9 600 ~0。

我们还在 CAudioManager 中加入了 GetVolume 和 SetVolume,其方法比较容易理解,这里不再占用篇幅逐一介绍。

5.4.4 DirectX 音量控制

我们希望读者会觉得“DirectMusic 音量教程”有用和有趣。正像前面所述,功能强大的 API 能使复杂的事情变简单,但也会把简单的事情变复杂。幸运的是,DirectMusic 音量控制相对于混音器 API 而言,并不太复杂性。关于混音器将在下节介绍。

继续学习前,请看看 Ch5p4_DmusicVolume 示例程序,它将演示样本音量和全局音量的控制。

5.5 用混音器 API 控制音量

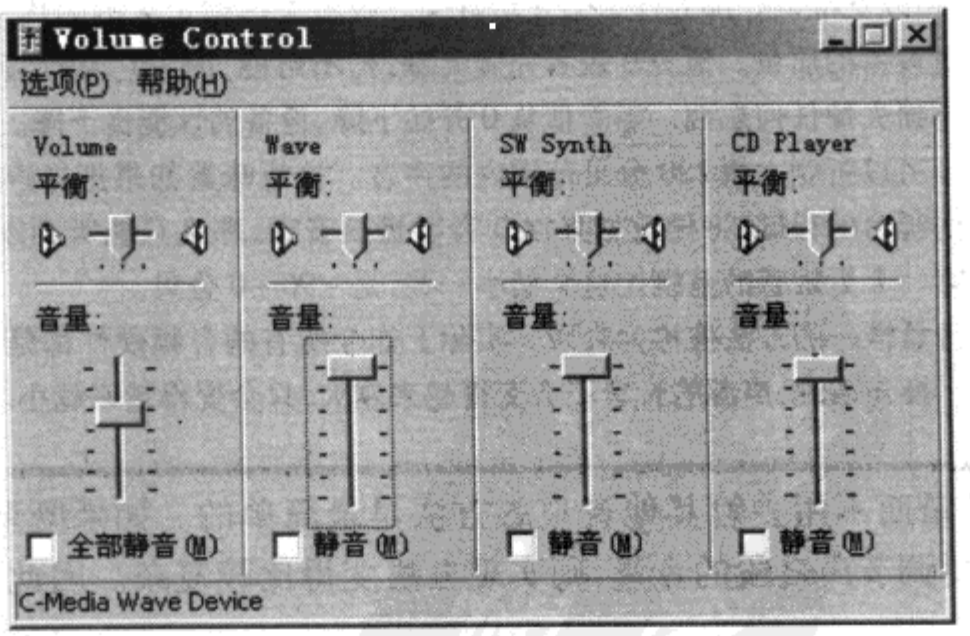


图 5.3 Windows 提供了一个 GUI 界面的混音器

到现在为止,已经学习的 DirectMusic 音量控制都是与系统的混音器音量有关的。混音器音量是指用户双击系统的扬声器图标并调节滑块所设置的音量。通常,滑块包括主音量输出、Wave 音频输出、CD 音频输出等,参见图 5.3。

用程序的方式来调节混音器的值是可行的,尽管这种方式要向下穿过好几个结构层和虚对象。本节将介绍具体操作方法。

提示

混音器控制系统中所有事物的音量调节,因此大多数的用户不喜欢在背后改变它们。举例说,如果你的游戏调节了主音量,它不仅改变了自身的音量,而且也改变了其他所有运行的应用程序。这种做法是很冒失的,因为玩家在玩游戏的同时,可能也在欣赏后台播放的 MP3 音乐,而他肯定会希望 MP3 的音量保持不变。

请不要认为混音器音量控制仅仅是一种简易的方法,因此而忽略了学习如何控制应用程序的音量输出。

5.5.1 混音器的体系结构

图 5.4 总结了混音器的体系结构。体系结构中有很多结构和虚拟对象,因此需要慢慢地理解。如果把应该控制 ID 的地方写成了混音器 ID,那么将花费很多时间在调试上。

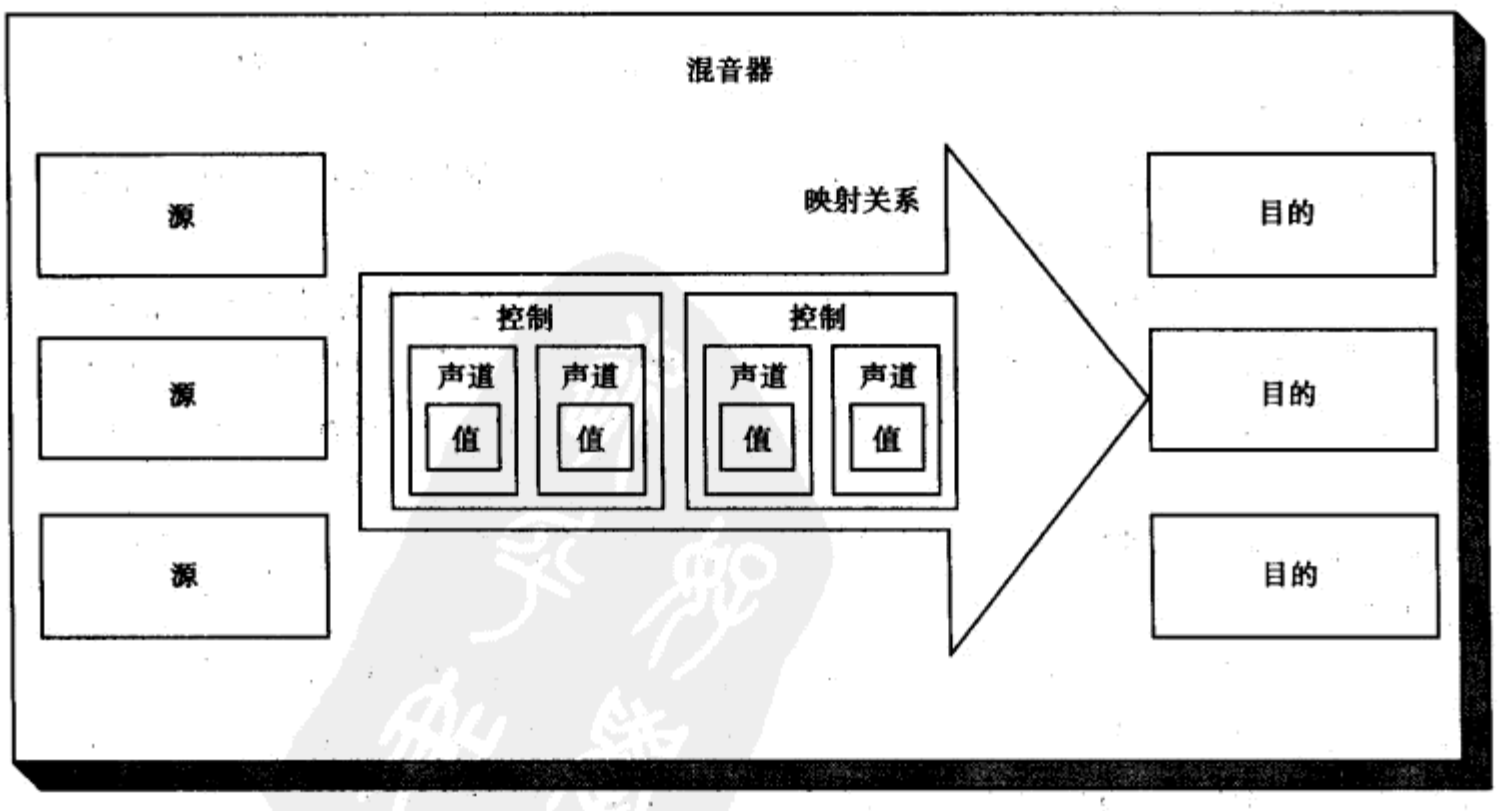


图 5.4 混音器 API 被分成了不同的虚拟对象

混音器体系结构的顶部是混音器。每个混音器包含几个源和目的。不能直接修改这些源和目的,但可以修改它们之间的映射关系。可以请求 API 得到任意两个源和目的之间映射关系的 ID,或者请求得到与某个特定目的(如主扬声器输出)相关联的所有源的映射关系 ID。

每个映射关系都有好几个不同类型的控制。除了常见的音量控制面板,还有更专业的控制,比如均衡器,这取决于所使用的音频硬件。值得注意的是 API 完全支持自定义的控制类型,即不是任何预定义的控制类型。

每个控制在映射关系上通常有相同数量的声道。比如,在立体声的映射关系上,每个控制通常都会有两个声道(左声道和右声道)。但也不一定总是这样,如很多声卡驱动程序的静音控制就只有一个声道,因此需要留意 API 返回的结构的变量。

每个声道有一个值。通常这个值是有符号或无符号整数,也可以是其他值。所给值的取值范围依赖于声卡。

5.5.2 改变混音器参数

前面简单介绍了混音器的体系结构,下面给出改变混音器某个特定值的步骤:

①调用 mixerOpen 函数获得一个混音器句柄。给第二个参数 uMxId 置 0,以告诉 Windows 打开第一个混音器设备。通常这种方式是可行的,因为第一个混音器设备也是惟一的混音器设备,但是如果用户的声卡不止一个,那么就需要调用 mixerGetNumDevs 和 mixerGetDevCaps 函数来遍历可用的混音器。

②指定要修改的映射关系,并指定它的源和目的。比如,对于 Wave 输出音量而言,源是 Wave 输出,目的是扬声器。

③将这些信息传递给 mixerGetLineInfo,它会返回一个映射关系 ID 和这个映射关系的声道数量。对于立体声 Wave 输出而言,声道数量就是 2。

④指定想要调节的映射关系中的控制。最简单的方法是请求映射关系提供特定类型(比如音量)的控制 ID。指定 MIXER_GETLINECONTROLSF_ONEBYTYPE 标记调用 mixerGetLineControls 函数,并将想要的控制类型放入 MIXERLINECONTROLS 结构的 dwControlType 变量中。

⑤得到一个描述控制的 MIXERCONTROL 结构,保存这个结构的 dwControlID 变量,它指示了想要的控制。

⑥指定所要的控制类型的细节信息,如音量控制为 MIXERCONTROLDETAILS _UNSIGNED,创建一个该类型的 value holder 结构数组。数组大小需要与映射关系中的声道数量相同。用控制 ID 和 value holder 结构数组对 MIXERCONTROLDETAILS 结构赋值。

⑦如果要查询控制,可以调用 mixerGetControlDetails。这个函数将用某个特定声道的当

前状态为每个值结构赋值，比如，数组中的第一个元素与声道 1 对应，第二个与声道 2 对应，以此类推。

⑧如果要对控制进行重新设置，要么必须对所有的值进行设置，要么一个都不设置。例如，不能要求 Windows 只设置双声道映射关系中的声道 1 的值。必须先查询所有的声道的值，改变设置，然后将新值（或没有没有改变的原值）存回系统。

⑨关闭在第 1 步中打开的混音器句柄。

5.5.3 构建 Cmixer 类

你可能从来没有想到音量控制会这么复杂，现在就来将这些复杂的事情封装到一个用户容易使用的 Cmixer 对象类中。这里，不打算一行代码一行代码地讲解 Cmixer，而是根据本书的设计思想来解释它的接口，读者可以根据具体代码和前面章节的内容来领会这个类是如何工作的。

所构建的 CMixer 类主要用于处理音量和控制静音，也可以把它推广用于处理其他控制类型，比如，耳机的增益控制，或重音和高音的控制。

CMixer 使用一个哑类 CMixerLine 来存储源和目的之间的关系。把 CMixerLine 称作哑类是因为它没有设置音量和静音的方法。换句话说，不能要求 CMixerLine 自己静音，而只能要求 Cmixer 把 CMixerLine 静音。

设计者也可把 CMixer 变得更像是 CMixerLine 工厂，也就是说，传给 CMixer 源和目的，然后如果这是一个合法的映射关系，那么 CMixer 返回一个 CMixerLine。接着就可以调用 CMixerLine 的函数来改变映射关系中的控制。

提示

在 Cmixer 对象类中没有提到的一件事是面板的保存。大多数的声卡提供了两个声道的音量：左扬声器和右扬声器。为了在设置这些值时保存面板，需要保证它们是相同的（或者，在后面明确地设置面板值）。本书把这些任务留给读者在需要的时候实现。

对于更复杂的混音器接口，这可能是设计者乐意选择的方式，但对于单一的音量控制，则宁愿为了简单而避开灵活性；在我的设计中，需要关注的对象类只有 CMixer 一个。

如果查看 CMixer 的类声明，会发现它包含了好几个 static const CMixerLine 变量。这些用来存储常见映射关系的源和目的，被修饰成常量。比如，如果要设置主音量，最好在 SetVolume 调用中使用内置 CMixer 的变量 Master，其代码如下所示：

```
//  设置主音量为 50%
m_Mixer. SetVolume( CMixer::Master, 32767);
```

当然,如果有兴趣,也可以自己指定源和目的映射关系:

```
//  设置电话(modem)音量为50%
m_Mixer. SetVolume(
  CMixerLine(
    MIXERLINE_COMPONENTTYPE_SRC_TELEPHONE,
    MIXERLINE_COMPONENTTYPE_DST_SPEAKERS
),
32767); //大约一半
```

提示

从网站上可以下载很多免费的混音器类来用,比如 www. codeguru. com 和 www. codeproject. com 网站。浏览这些网站,你可能会得到一些好东西。

代码中创建了一个临时的对象类 CMixerLine 来指定一个电话(源)和一个扬声器(目的)。

Ch5p5_MixerVolume 示例程序演示了如何使用 CMixer,并在随书的 CD 光盘中包含了两个关于混音器 API 的链接。

5.6 本章小结

本章介绍了很多独立的主题和 API,音频引擎的功能也得到不断增强。现在音频引擎已经具备了最主要的功能,虽然它还缺少一些功能,但已经可以集成到简单的游戏中去了。

在接下来的一章中,将学习如何进一步扩展音频引擎的功能,特别是播放音乐、动态音乐、3D 声效等。

PDG

第 6 章

播放 MIDI

前面的章节讲授如何加载、播放和管理声效,现在把注意力转移到游戏音频的另一个重要内容:背景音乐。

在本章中,主要学习关于使用 MIDI 文件的核心技术。主要内容是关于静态音乐的播放,也就是所播放的音乐不会改变。在本书第二部分中,将学习如何处理动态音乐。

DirectMusic 采用相同的方式来对待 WAV 文件和 MIDI 文件:首先由 Loader 对象把 MIDI 文件或 WAV 文件加载到声效段中,然后把声效段数据传给演奏对象的 Play 函数进行播放。

6.1 MIDI 工作原理

MIDI 是乐器数字接口(Musical Instrument Digital Interface)首字母的缩写,最早出现于 20 世纪 80 年代,如今,MIDI 已经成为电子乐器间进行通信的事实上的标准。

可以把 MIDI 看作是一个简单的网络协议。本质上讲,就是用 MIDI 线缆把所有乐器硬件(合成器、打击乐器、计算机等)都连接起来,并为每个硬件分配一个惟一的通道 ID 见图 6.1。例如,你的合成器可能是通道 2,而你的打击乐器可能是通道 4。现在,当你在合成器上敲击一个键时,通道 2 发出一个“音符开”的消息。当你在打击乐器上敲击鼓时,通道 4 就会发出一个“音符开”的消息。MIDI 支持 16 个不同的通道。

计算机软件(或一个高级的合成器)能够监听任意一个或所有的 MIDI 通道,并记录“音符开”事件。通过在恰当的通道上发送消息可以播放音乐,或者可以将接收到的音符序列存储到 MIDI 文件中。

音频剪辑

播放音频剪辑 6.1 和 6.2,可以听到不同的 MIDI 音乐例子。

“音符开”消息本身包含几个不同的字段,有的表示所播放音调的数字(比如中央 C 音调,或 C4,数字越高表示音调越高),还有的表示可选的音符音量、持续时间等。对于鼓乐器,“音符”实际上是鼓的类型,当鼓合成器接收到一个 C4 音符时,它播放较高的小鼓声音。对于打击乐器,越高的音符数字并不表示越尖的鼓声,而是表示不同鼓声,有点像 ASCII 字符集。

MIDI 定义了很多不同类型的消息。“音符开”消息是最重要的一个,还有其他一些消

息被用来告诉设备改变它们的音量和执行特效(比如颤音或变调)。

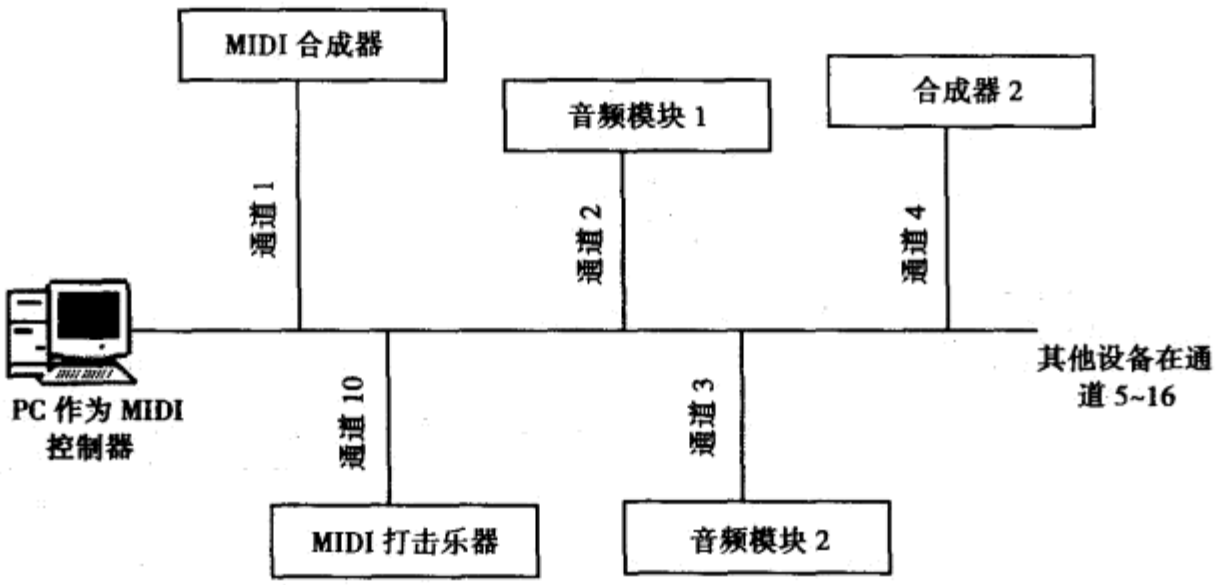


图 6.1 专业 MIDI 配置可能有好几个不同的硬件设备,它们被分配不同的 MIDI 通道

MIDI 通道与音轨不一样。音序器软件(比如 DirectMusic Producer 或者 Cakewalk Pro Audio)使用音轨的概念。虽然只有 16 个 MIDI 通道,但可以有几百个音轨。即在任何时刻,至多只能拥有 16 个活动的通道,但这 16 个通道,还可以分成更多的音轨。比如,一些音轨可能被静音,因为它们包含了不同的声效。但这并不表示只能同时播放 16 个音符,每个通道与一个设备或乐器对应,很多设备或乐器都允许同时播放几个音符(被称为复音)。

在 DirectMusic 中,MIDI 消息通常不会从 MIDI 端口发出,然后通过 MIDI 线缆到达外部的合成器,而是被直接发往 DirectMusic 内部的合成器,该合成器通过软件合成声音并输出到扬声器。这种方式对于视频游戏有很大好处,因为不需要花几千元购买合成器和设备,才能把 MIDI 文件变成声音,所需要的仅仅是 Windows 附带的标准 DirectMusic 软件合成器。

6.2 类的变更

在播放 MIDI 音乐前,需要决定如何让 MIDI 文件适合音频引擎的结构。现在,音频引擎使用 CSound 类来负责声效。这种方式虽然可行,但并不是最好的。根据 C ++ 设计原理,对代码做些小的调整,把对象类 CSound 作为一个基类,且有 3 个新的继承类:CDirectMusicSegment、CSoundEffect 和 CMIDIMusic。图 6.2 表示了这种新的类层次结构。

这里,CSound 类被削减了一些代码。在前面章节中,它包括 DirectMusic 段、音量和其他所有相关信息。但现在,它实际上仅仅是作为一个抽象的基类。这是因为在类继承关系中,把 Csound 作为所有输出到扬声器的音频对象的基类,既包括声效和 MIDI 音乐(使用 DirectMusic 段的对象),也包括不需要 DirectMusic 参与处理的对象,像 MP3 或者 Ogg Vorbis 音乐文件。

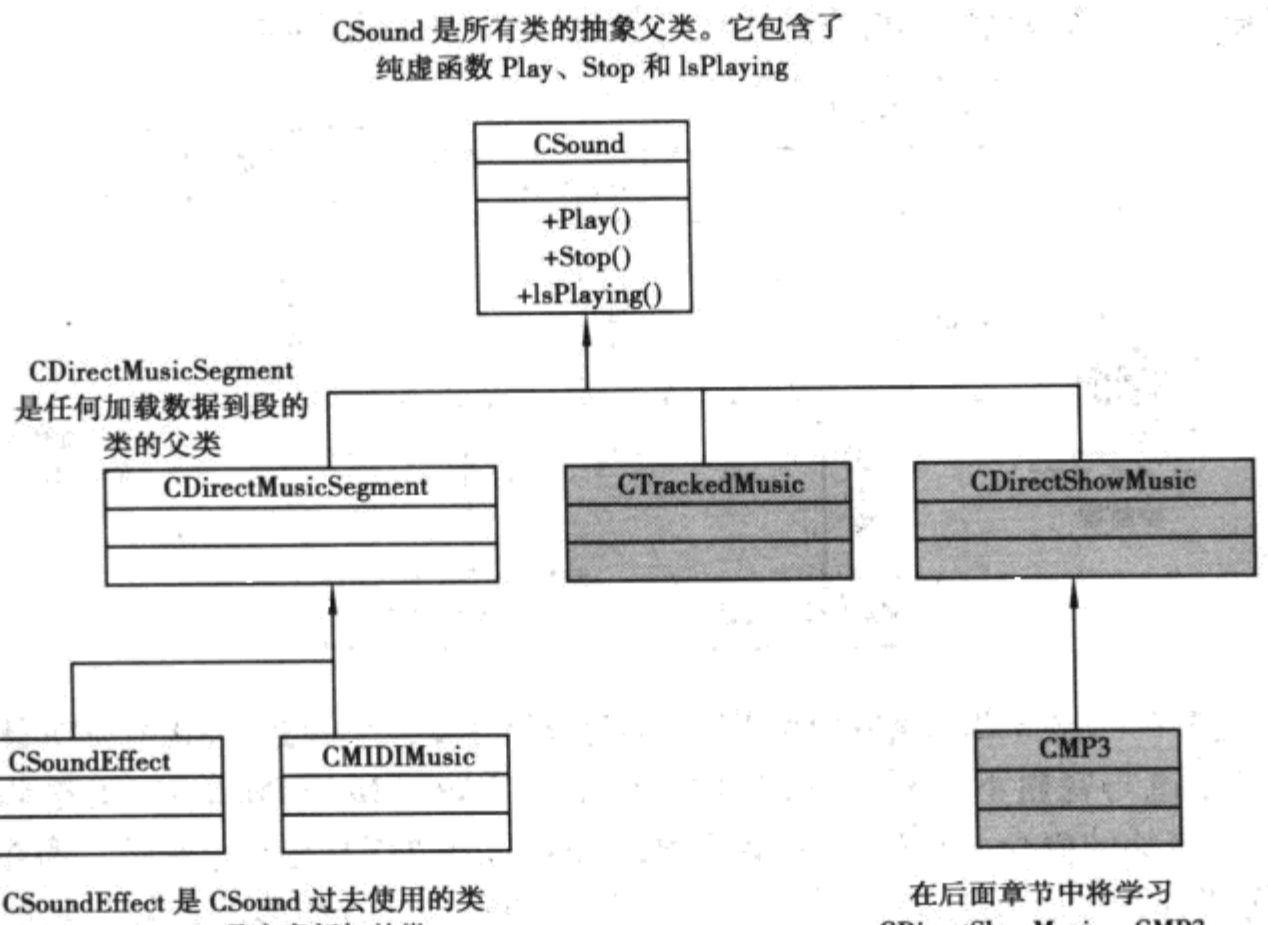


图 6.2 音频引擎的类层次图

CSound 有 3 个纯虚函数:Play、Stop 和 IsPlaying,以及 2 个虚函数:GetVolume 和 SetVolume。这 5 个函数是播放任何类型的音频所需要的最小接口。无论是 MP3、WAV 文件还是 DirectMusic 编曲的动态音乐块,至少都需要播放、停止、以及检查是否正在播放和调节音量。

从 CSound 继承的子类 CDirectMusicSegment 表示一个 DirectMusic 段,它包含了过去在 CSound 中的功能。它有 DirectMusic 段指针,并且在 Play、Stop 和 IsPlaying 中使用这个段指针。如果以后要对段做更多的操作(不管是音乐段还是声效段),可以把那些函数都放在这个类中。

从 CDirectMusicSegment 继承的另外两个子类是 CSoundEffect 和 CMIDIMusic。它们只针对声效处理或者只针对 MIDI 音乐处理(比如设置音乐节拍)。

把声效和音乐分成独立的类是愚蠢的做法,特别是因为DirectMusic 把WAV 文件和 MIDI 文件都加载到相同的段对象中,虽然这使得你可以用相同的方式对待 WAV 文件和 MIDI 文件。但是,不能因为可以这样做就意味着一定这样做。我们有可能需要处理一些对音乐文件是需要的、但对声效文件却不需要的特殊情况。比如音乐节拍,它在处理 MIDI 音乐时需要考虑,而在处理 WAV 声效时却不需要。格式可以互换,WAV 文件和 MIDI 文件都可以用

于声效和音乐,但是“背景音乐”和“声效”之间仍然有概念上的差别。如果实际上不想区分这种差别,可以只使用 CDirectMusicSegment 类,而不用 CSoundEffect 和 CMIDIMusic 子类。

音频引擎的类继承关系的其余部分保持不变。管理器仍然检查整个操作,而其他的类(CWAVFile、CMixer 等)也都没有被修改。

6.3 加载和播放 MIDI

本章开头曾提到,DirectMusic 尝试用相同的方式来处理 MIDI 音乐和 WAV 文件,它们都通过加载器加载到段中。令人高兴的是:从硬盘、内存或者资源中加载 MIDI 歌曲到一个段中的代码与加载 WAV 文件的代码是完全一样的。

6.3.1 增强音频管理器

音频管理器的加载程序的功能需要得到增强,以满足加载 MIDI 的要求。音频管理器允许通过 LoadSound 函数加载一个 WAV 文件。比较好的设计方法是创建 3 个新的 LoadMIDI 对象类来加载 MIDI 文件,一个用于从硬盘加载,一个用于从内存中加载,另一个用于从资源加载。

不能简单复制 LoadSound 程序并改名,更不要复制较大的代码块,因为这样做会增加需要处理的代码量。较好的解决方案是提炼出公用的代码并放入如下 3 个核心函数中,然后让 LoadSound 和 LoadMIDI 程序调用这些函数。这三个核心函数是 LoadSegmentFromDisk、LoadSegmentFromMemory 和 LoadSegmentFromResource,它们具有不同的参数。举例说明其中的一个,下面是 LoadSegmentFromDisk 代码段:

```
void CAudioManager::LoadSegmentFromDisk(CDirectMusicSegment * dest, std::string filename)
{
    HRESULT hr;
    // 把文件名转换成宽字符串
    WCHAR widefilename[MAX_PATH];
    DXUtil_ConvertGenericStringToWideCch(widefilename, filename.c_str(), MAX_PATH);

    // 让加载其加载文件
    hr = m_Loader -> LoadObjectFromFile(
        CLSID_DirectMusicSegment,
```

```
        IID_IDirectMusicSegment8,
        widefilename,
        (void * *) &dest -> m_Segment);
    ThrowIfFailed(hr, "CAudioManager::LoadSound(std::string filename):
        LoadObjectFromFile failed.");
}
```

这段代码几乎包含了以前磁盘加载 LoadSound 函数的所有内容,所缺少的内容仅仅是类的实例化。LoadSegmentFromDisk 传递了一个对象指针,以取代创建一个新类对象。这是因为在设计中,LoadSound 创建 CSoundEffect 对象并传给 LoadSegmentFromDisk,而 LoadMIDI 则创建 CMIDIMusic 对象。由于对象的多样性,LoadSegmentFromDisk 并不知道 CMIDIMusic 和 CSoundEffect 对象间的区别,LoadSegmentFromDisk 使用它们共有的父类 CDirectMusicSegment。下面是修改后的新版本 LoadSound 函数:

```
CSoundPtr CAudioManager::LoadSound(std::string filename)
{
    CSoundEffect *snd = new CSoundEffect(this);
    LoadSegmentFromDisk(snd, filename);
    return(CSoundPtr(snd));
}
```

读者可以看到,LoadSound 现在只负责创建对象,它调用 LoadSegmentFromDisk 完成底层处理,然后返回一个新的 CSoundPtr。这与 LoadMIDI 本质上是相同的:

```
CSoundPtr CAudioManager::LoadMIDI(std::string filename)
{
    CMIDIMusic *music = new CMIDIMusic(this);
    LoadSegmentFromDisk(music, filename);
    return(CSoundPtr(music));
}
```

LoadMIDI 创建的不是 CSoundEffect 对象,而是 CMIDIMusic 对象,然后做与 LoadSound 相同的事情。

另外两个 LoadSound 函数和 LoadMIDI 函数的功能也是相似的,它们只是创建适当的对象,然后传给 LoadSegmentFromMemory 或 LoadSegmentFromResource 进行处理。

6.3.2 获得 CMIDIMusic 指针

当加载一个 MIDI 文件后,如何才能获得特定功能的 CMIDIMusic 呢? 比如,可能需要

改变音乐节拍等等,虽然 SetTempo 函数在 CMIDIMusic 类中,但现在只知道指向 CSound 的指针。

为此,可以使用运行时的类型信息。RTTI 被设计为用来处理这种情况:已有一个指向父类的指针,但需要知道它实际的子类。下面的例子将说明如何从 CsoundPtr 获得一个 CMIDIMusic 指针:

```
CSoundPtr snd = g_AudioMgr. LoadMIDI("test. mid");
CMIDIMusic * music = dynamic_cast < CMIDIMusic * > (snd. Get());
If (music ! = NULL) {
  // snd 实际上是一个 CMIDIMusic 指针! 做相关的事情!
}
Else {
  // snd 不是一个 CMIDIMusic 指针!
}
```

注意,关键字 dynamic_cast 将一个指针转换成不同的类型。这里,代码调用 snd. Get 得到指针,然后尝试将指针转换成 CMIDIMusic 指针。如果 dynamic_cast 正常工作,转换后的指针将存入 music 中(music 和 snd 指向相同的内存地址,它们只是类型不同)。如果因某种原因 dynamic_cast 没有成功转换指针(如传递 CSoundEffect 指针),它将把 music 赋值为 NULL。

通过这种方式,在得到 CSound 或 CDirectMusicSegment 指针时,可以得到 CMIDIMusic 或 CSoundEffect 的函数。

6.4 改变音乐节拍

处理 MIDI 文件时可以做的一件事情(处理 MP3 或 WAV 时则不行)是调节播放的音乐节拍。Tetris 大量使用音乐节拍变化来制造悬念,当游戏空间几乎要满的时候,音乐的播放加速,而当(或者如果)局面重新控制住时,音乐的播放又回到正常的速度。这个简单的技巧提高了游戏音效的质量。

6.4.1 记录节拍与演奏节拍

DirectMusic 有两种不同的改变音乐节拍的方法:可以直接改变保存在段中的音乐节拍记录参数;也可以屏蔽掉音乐节拍记录,让演奏对象来改变所播放音乐段的演奏节拍。

在改变音乐段的演奏节拍后,如果以后要查询音乐段的记录节拍,仍然可以得到实际的音乐节拍。比如,如果加载一个 MIDI 文件,其音乐节拍本来是每分钟 109 拍,然后把演奏

对象的音乐演奏节拍变成了每分钟 200 拍,当查询段的音乐节拍时,将仍然得到 109 拍的值(段的音乐节拍确实是 109,只是因为演奏对象的设置而用每分钟 200 拍播放)。

如果想在改变音乐节拍时让所有正在播放的音乐都加速播放,而不只是一个音乐段,则需要设置演奏对象的演奏节拍。

改变音乐节拍的两种方式的代码都相当简单,下面是 CMIDIMusic::SetTempo 的代码:

```
void CMIDIMusic::SetTempo(int beats_per_minute)
{
  //第 1 步:设置音乐节拍记录参数
  DMUS_TEMPO_PARAM Tempo;
  Tempo.dblTempo = beats_per_minute;
  ThrowIfFailed(m_Segment -> SetParam(GUID_TempoParam, 0xFFFF, 0, 0, &Tempo),
    "CMIDIMusic::SetTempo: SetParam failed!");

  //第 2 步:设置演奏对象的音乐节拍
  DMUS_TEMPO_PMSG * pTempo;
  ThrowIfFailed(
    m_Manager -> GetPerformance() -> AllocPMsg(
      sizeof(DMUS_TEMPO_PMSG),
      (DMUS_PMSG **)&pTempo),
    "CMIDIMusic::SetTempo: couldn't allocate PMsg!"
  );
  // 提示音乐节拍事件
  ZeroMemory(pTempo, sizeof(DMUS_TEMPO_PMSG));
  pTempo -> dwSize = sizeof(DMUS_TEMPO_PMSG);
  pTempo -> dblTempo = beats_per_minute;
  pTempo -> dwFlags = DMUS_PMSGF_REFTIME;
  pTempo -> dwType = DMUS_PMSGT_TEMPO;

  ThrowIfFailed(
    m_Manager -> GetPerformance() -> SendPMsg((DMUS_PMSG *)pTempo),
    "couldn't send PMsg!"
  );
}
```

这段代码根据 DirectX 文档中所给出的示例代码而编写。可参考 DirectX Audio\Using DirectX Audio\Playing Sounds\Timing\Changing the Tempo。

第一步是设置音乐节拍记录参数为给定的值 beats_per_minute。程序代码对 DMUS_TEMPO_PARAM 结构赋值,然后把该结构传给段接口的 SetParam 函数以设置音乐节拍的记录参数。DMUS_TEMPO_PARAM 结构只有一个变量 dblTempo,它记录你想设置的音乐节拍。

第二步涉及的操作要稍微多些。首先,程序代码必须分配一个新的 PMsg 结构,并把它存在 pTempo 中。分配完成以后,代码对结构赋值。在这儿最重要的变量是 dblTempo,它被用来设置需要改变的演奏对象的演奏节拍。

注意,在调用 ZeroMemory 时,结构中的变量将被自动置成 0。但变量 dwFlags 不能为 0;它必须被置成标记 DMUS_PMSGF_REFTIME 或 DMUS_PMSGF_MUSICTIME。不用关心选哪一个标记,因为 rtTime 变量和 mtTime 变量都由 ZeorMemory 置成 0,于是可挑选一个字母短的。

同样,dwType 变量也不能为 0,它必须赋值为 DMUS_PMSGT_TEMPO,以便让 DirectMusic 知道它得到的是音乐节拍事件。

最后,当结构被赋值完以后,代码调用 SendPMsg,将发送 pTempo 消息,并让 Directmusic 改变音乐节拍。

6.4.2 获得音乐节拍

获得音乐节拍比设置它简单得多。获得音乐节拍只需传递 DMUS_TEMPO_PARAM 结构给段的 GetParam 对象,如下所示:

```
int CMIDIMusic::GetTempo()
{
  DMUS_TEMPO_PARAM Tempo;
  m_Segment -> GetParam(GUID_TempoParam, 0xFFFF, 0, 0, NULL, &Tempo);
  return(Tempo.dblTempo);
}
```

它没有什么复杂操作,只是把参数 GUID_TempoParam 和存放结果的结构 Tempo 传给 GetParam 对象,然后就可以得到想要的信息。不用关心 GUID_TempoParam 和 Tempo 之间的参数。如果想进一步了解它们,可以查看 DirectX 文档中的 GetParam。

6.4.3 音乐节拍缩放因子

DirectMusic 还有另外一种音乐节拍控制方法,称为主音乐节拍。当主音乐节拍作为演

奏对象音乐节拍的缩放因子时,它并不是真正的音乐节拍。它的默认值是1,可以根据需要改变它,如果赋值在0~1之间,将放慢演奏对象的播放速度,如果赋值大于1,将加快播放速度。DMUS_MSTERTEMPO_MIN 常量和 DMUS_MSTERTEMPO_MAX 常量定义了其最小值和最大值。

主音乐节拍对于 Tetris 风格的音乐加速播放很有用,因为不需要知道原始的音乐节拍就可以加速播放一首歌曲。所要做的就是把主音乐节拍调高到1以上来加速播放,然后重新设置成1恢复正常的音乐节拍。

获得和设置主音乐节拍的值比较简单。本书在 CAudioManager 中加入了两个新的函数来完成这些工作。首先是 SetMasterTempoScaleFactor,其代码为:

```
void CAudioManager::SetMasterTempoScaleFactor(float value)
{
  if (NULL == m_Performance) return;
  ThrowIfFailed(m_Performance -> SetGlobalParam(GUID_PerfMasterTempo, &value,
    sizeof(value)),
    "CAudioManager::SetMasterTempo: SetGlobalParam failed.");
}
```

这种方式与设置主音量的方式相同,在调用 SetGlobalParam 函数时,传递你想设置的 GUID、新值以及新值的字节长度,它没有复杂操作。

然后是 GetMasterTempoScaleFactor,其代码如下:

```
float CAudioManager::GetMasterTempoScaleFactor()
{
  if (NULL == m_Performance) return(0.0f);
  float tempo = 0.0f;
  ThrowIfFailed(m_Performance -> GetGlobalParam(GUID_PerfMasterTempo, &tempo,
    sizeof(tempo)),
    "CAudioManager::GetMasterTempo: GetGlobalParam failed.");
  return(tempo);
}
```

它调用 GetGlobalParam 函数,提供的参数包括想查询的 GUID、存储值的变量以及变量长度。DirectMusic 将返回所需要的任何一个全局参数。

当不关心歌曲的音乐节拍,而只是想加快播放时,这两种简单方法都非常有用而且行之有效。

6.5 检测音乐节拍

处理 MIDI 时,可以检测音乐的拍子并对其进行一些操作。比如,可以在游戏中让一个人随着背景音乐跳舞,或者让闪光灯和其他东西随着拍子闪烁。

> **提示**
>
> 从拍子产生到拍子被代码检测有一个延时(20 ~ 100 ms)。这个延时对人而言是察觉不到的,因此,当你想把图片跟音乐同步时,可以不考虑延时的影响。但如果要实现一些高级功能(与其他机器同步或者合成 DirectMusic 的拍子),则可能需要计算延时。关于如何做,可参见 DirectX 文档。

6.5.1 通知的概念

检测节拍需要一段代码。DirectMusic 没有像“isPlayingBeat”的函数可以调用,取而代之的是用另一种不同的方式进行交互:使用被称为“通知”的事件。

当感兴趣的事情发生时(比如一个拍子,或者一个段循环,或者其他事情),DirectMusic 将该事件加入它内部的队列中(有点像 Windows)。可以使用演奏对象的 GetNotificationPMsg 函数从队列中取出这些事件。

在这样做之前,必须事先告诉 DirectMusic 你所感兴趣的事件类型,可以调用演奏对象的 AddNotificationType 来完成。每个事件是一个 GUID;将感兴趣的事件传递给 AddNotificationType 的 GUID,见表 6.1。

> **提示**
>
> 请记住以前见过的 PMsg:当你设置音乐节拍时,你传给演奏对象一个 DMUS_TEMPO_PMSG。而现在你是在接收信息。

图 6.3 描述了使用 DirectMusic 执行事件的基本流程。先在初始化程序中调用 AddNotificationType,然后可以周期性的调用 GetNotificationPMsg(例如在主游戏循环中)。

如果 GetNotificationPMsg 得到一个消息,则返回 S_OK;如果没有得到消息,则返回 S_FALSE。因此,可以调用 GetNotificationPMsg 来处理从当前到前一个时刻所发生的所有消息。其代码段如下:

```
DMUS_NOTIFICATION_PMSG * pPmsg;
while (m_Performance -> GetNotificationPMsg(&pPmsg) == S_OK) {
  // 用 pPmsg 做感兴趣的操作
```

```
// 操作完后,释放它
m_Performance -> FreePMsg( ( DMUS_PMSG * ) pPmsg) ;
}
```

表 6.1 “通知”中 GUID 取值所对应的事件

GUID	事件
GUID_NOTIFICATION_CHORD	发生旋律改变。这主要用于告诉你 DirectMusic 作曲器准备改变正在播放的旋律
GUID_NOTIFICATION_COMMAND	发生一个命令事件
GUID_NOTIFICATION_MEASUREANDBEAT	产生拍子时,dwField1 的值为 0 表示一个新的测量,其他值则表示测量中的拍子
GUID_NOTIFICATION_PERFORMANCE	发生了一个执行事件。你可以查看 dwNotificationOption 来判断具体是什么事件
GUID_NOTIFICATION_RECOMPOSE	一个音轨被 DirectMusic 作曲器重新编写
GUID_NOTIFICATION_SEGMENT	发生了一个段事件。你可以查看 dwNotificationOption 来判断具体是什么事件

GetNotificationPMsg 有一个指针参数,它指向待赋值的 DMUS_NOTIFICATION_PMSG 结构。DMUS_NOTIFICATION_PMSG 结构包含关于消息的所有信息,包括类型 guidNotificationType、可选子类型 dwNotificationOption,以及任意可选参数 dwField1 和 dwField2。

注意

当完成了从 GetNotificationPMsg 得到通知事件的操作,必须调用演奏对象的 FreePMsg 方法来删除它。如果没有这样做造成内存漏洞!

6.5.2 将通知引入音频引擎

前面介绍了 DirectMusic 中通知/事件的基本模式,现在将它应用集成到音频引擎中。需要考虑的事情是把事件告诉给音频引擎的代码。不同的游戏期望检测不同的事件,并且用不同的方式响应它们。音频引擎需要通过一种简单方式告诉游戏代码发生了什么事情。

在C语言中,很多程序员都使用函数指针来解决这个问题。客户端代码提供在某个时刻(比如当一个拍子发生时)要调用的函数地址,而音频框架则在该时刻调用这个函数指针。

C++程序员更深入一步,他们使用一个类来代替函数指针。这个类包含各个可能发生事件类型的虚拟函数。音频引擎有一个指向该类的指针,当时间到时,音频引擎使用该指针来调用类。

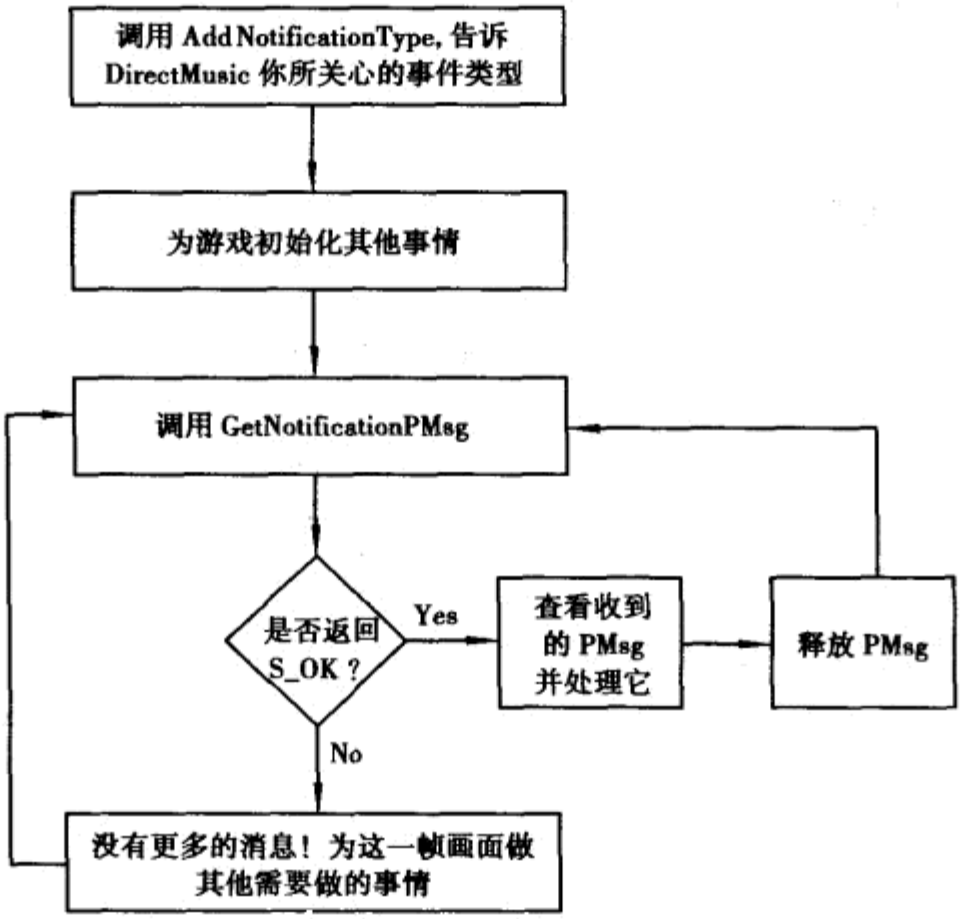


图6.3 使用DirectMusic演奏对象事件的基本处理流程图

例如,把这个类定义为CNotificationHandler。它包含了用于需要响应的各种事件的虚拟函数:

```
class CNotificationHandler
{
public:
    CNotificationHandler( ) { }
    virtual ~CNotificationHandler( ) { }
    //这个函数在拍子发生时调用
    virtual void OnBeat( DMUS_NOTIFICATION_PMSG &msg) { }
    //这个函数在一个新的测量被计数时调用
```

```
    virtual void OnMeasure(DMUS_NOTIFICATION_PMSG &msg) { }
    //等等……
};
```

注意,所有这些函数都是虚拟函数,它们没有进行任何操作。现在假设音频引擎有一个指向该类的指针叫做 m_NotificationHandler。m_NotificationHandler 的默认值是 NULL,游戏代码可以改变它。当一个事件发生时,音频引擎使用 m_NotificationHandler 指针来调用适当的方法:

```
void CAudioManager::DispatchNotificationMessages()
{
  DMUS_NOTIFICATION_PMSG * pPmsg;
  while (m_NotificationHandler &&
        m_Performance->GetNotificationPMsg(&pPmsg) == S_OK)
  {
    if (pPmsg->guidNotificationType == GUID_NOTIFICATION_MEASUREANDBEAT) {
      // 如果它是一个拍子,就调用 OnBeat,否则调用 OnMeasure。
      if (pPmsg->dwField1 == 0) {
        m_NotificationHandler->OnMeasure( * pPmsg);
      }
      else {
        m_NotificationHandler->OnBeat( * pPmsg);
      }
    }
    //不要忘记在操作完以后释放消息
    m_Performance->FreePMsg((DMUS_PMSG * )pPmsg);
  } // while 循环结束
}//函数结束
```

这种设计的好处是游戏代码可以从 CNotificationHandler 继承得到一个新子类,这个子类重载了虚拟函数。然后代码可以设置 m_NotificationHandler 指针指向该子类的实例,于是可以用期望的操作来响应事件。

这就是我所编写的音频管理器。管理器对象有两个新的祖先类:SetNotificationHandler 和 GetNotificationHandler,这两个类使读者可以设置和得到 m_NotificationHandler 指针。游戏周期性的调用 DispatchNotificationMessages 对象,而 DispatchNotificationMessages 使用 m_NotificationHandler 指针,通过调用 OnMeasure、OnBeat 以及其他函数来分发事件。

本章的示例程序演示了这个系统是如何工作的。在 Ch6p1_MIDIPlayback 中有一个称为 CCh6p1StupidNotificationHandler 类的定义。这个类继承 CNotificationHandler,但它有不同的实现方法,其代码如下:

```
class CCh6p1StupidNotificationHandler : public CNotificationHandler
{
  virtual void OnBeat(DMUS_NOTIFICATION_PMSG &msg) {
  cout << "..Beat!..";
  }
  virtual void OnMeasure(DMUS_NOTIFICATION_PMSG &msg) {
    cout << endl << "Measure!..";
  }
};
```

正如读者看到的,这个类的行为只是简单地输出消息。

6.5.3 检测多线程事件

如果不想始终都调用音频管理器的 DispatchNotificationMessages,还有另外一种选择:多线程。可以创建一个独立的线程,其惟一工作就是检测和响应 DirectMusic 事件。

肯定没有人想在这个独立线程中不断循环,因为那样会浪费掉宝贵的 CPU 资源。可行的做法是让这个线程休眠,直到接受到一个消息,分发这个消息,然后线程又回到休眠状态,直到传来下一个消息。

> **注意**
>
> 在创建事件时,请确保设置成自动重置,或者在被通知以后手工重置。否则,在得到第一个消息以后,将永远不能返回睡眠状态。
>
> 为了能自动重置事件,可以指定 CreateEvent 的第二个参数为 false。查看 Win 32 API 文档可获得关于 CreateEvent 更多的信息。

DirectMusic 提供了实现这种线程的方法。通过给 Windows 事件一个句柄,然后当产生一个新消息时会唤醒这个事件。实现的方法是使用 Win32 API 的 Create Event 函数创建事件,然后将事件传给演奏对象的 SetNotificationHandler 函数,从而告诉 DirectMusic:希望它使用这个事件。

现在可以编写自己的线程了,使用 WaitForSingleObject,传给它与传给 DirectMusic 相同的事件句柄,这样它就会自动休眠直到 DirectMusic 唤醒这个事件。当它醒来以后,线程可以使用 GetNotificationPMsg 获得消息并做相应的事情,然后调用 FreePMsg 释放掉消息,并休眠直到下一个事件发生。

DirectX 帮助文档中有这个代码的例子,可以在 DirectX Audio\Using DirectX Audio\ Pla-

ying Sounds\Notification and Event Handling 中找到(包括了它如何工作的更详细解释)。

音频引擎创建了一个事件并将其告诉 DirectMusic,可以通过 CAudioManager 的 GetNotificationEvent 来访问这个事件;然后,用 WaitForSingleObject 编写代码,直到得到一个消息才调用 DispatchNotificationMessages。

本书没有提供包含多线程的示例代码,但是把书中所讲的片断合起来应该是相当简单的,我把它留给读者作练习。

6.6 用 Windows MCI 播放 MIDI

如果不想使用 DirectMusic 播放 MIDI,还可以使用 Windows MCI 来播放它们。注意,Windows MCI 并没有提供对 MIDI 文件的精确控制,比如说,不能将它们与其他事情同步,并且可能不能播放某些 MIDI 文件。详细内容请察看 Platform SDK\Graphics and Multimedia Services\Multimedia Audio\Musical Instrument Digital Interface (MIDI)\About MIDI\Media Control Interface (MCI)中的 Windows Platform SDK 文档。

第三种播放 MIDI 文件的方法是:使用低层的 MIDI 输入和输出例程。这些 API 例程通常被多数音序器程序用来在低层记录、播放和管理 MIDI 数据。这里没有不能做的事情,但大部分事情的实现都相当麻烦,而且必须写大量的代码。多数游戏都不需要涉及低层的 MIDI 接口,需要高级控制或 MIDI 同步时,最好使用 DirectMusic;如果只想简单的播放 MIDI 而又不受 DirectX 约束,最好使用 MCI。

6.7 本章小结

本章完成了几件事。最简单的事是将 MIDI 文件和 WAV 文件都加载到段中,并用相同的方法来播放。同时,对音频引擎做了一些重组(为后续章节作准备),并讲解了如何发送和接收 DirectMusic 的消息。

下一章将学习如何播放其他格式的音频文件(包括 Ogg Vorbis 和 MP3)。此后,在学习动态音乐时,将回到声效段和演奏接口,并更透彻地学习它们。

第 7 章

播放 MP3 和 WMA

现在开始学习除 MIDI 以外的其他音乐格式的读取和播放。本章学习当前的两种主流格式:MP3(MPEG Layer 3)和 WMA(Windows Media Audio)。后续章节还将学习 Ogg Vorbis 格式、CD 音频和音轨音乐。首先需要知道如何播放 MP3。

7.1 播放 MP3

假定读者已经知道 MP3 的基础知识,比如 MP3 表示 MPEG Layer 3,而 MPEG 表示运动图像专家组(Motion Picture Experts Group,这个组织还提供 DVD 的视频算法)等等。我们需要了解的是 MP3 如何工作的,以及如何编写代码来播放 MP3。

7.1.1 MP3 工作原理

音频剪辑

播放音频剪辑 7.1、7.2 和 7.3,你可以感受 MP3 是如何压缩声音的。

MP3 的神奇之处在于它的音质跟 WAV 文件差不多但存储容量却小很多。一个典型的 WAV 文件根据记录选项和歌曲长度,需要占用 20 ~ 40 MB 的空间,而 MP3 通常只有这个大小的 1/10,并且它们听起来效果基本上是一样的。是魔术吗? 不是,只是进行了心理声学压缩。

MP3 能够在保持音质基本相同的情况下,使歌曲存储容量只有 WAV 文件的 1/10,其原因在于人的听觉系统并不是特别灵敏。一首歌是各种频率和振幅声波的集合,而人的听觉并不能同时分辨出所有不同的声波。一些声波的频率太高或者太低,而一些声波被音量大的所掩盖。MP3 压缩的原则是"如果听不到这种声波,又何必对它进行编码呢?"

一个尖锐的碰撞声将掩盖背景中的吉他声,因此当发出碰撞声时,就没有必要花费空间来编码吉他声的声波。此外,也没有必要花费上千个字节来记录超出了人耳听力范围的高音声波。家里的小狗可能更喜欢 CD 音轨而不是 MP3,但你的耳朵并不能听到丢掉的高频率声波,因此 MP3 的音质已经足够了。

7.1.2 MP3 文件格式

令人奇怪的是 MP3 文件没有文件头部,它不像 RIFF 文件有一个嵌入结构并在文件开

头有几个头部字节。取而代之,MP3 文件包含很多帧,而每个帧是一个自包含的歌曲片断。每个帧都有它自己的头部,这个头部记录它的比特率、MPEG 版本(目前版本 3 是最常用的,也有用版本 1 和版本 2 的)以及其他信息。

图 7.1 描述了一个帧头部的细节信息。正如所看到的,整个帧头部有 32 位长,信息被存放到这 32 位里,它并没有考虑字节的边界,必须使用位操作来提取信息片断(比如,比特率被存放到了第 3 个字节的高位)。MP3 结构这样设计是为了节省空间,专家组希望 MP3 尽可能更小,这也意味着确保每一位都保存有用的信息。

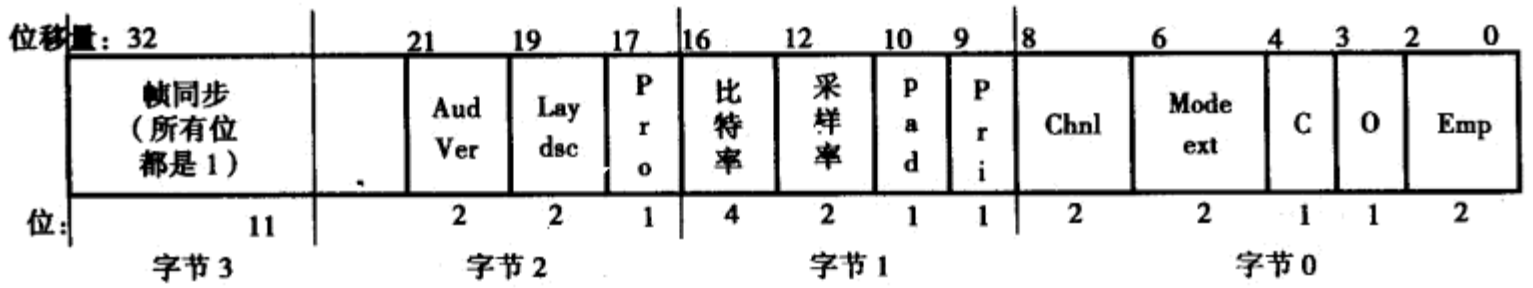


图 7.1 帧头的 4 个字节的内容

这里,不再描述每个信息片断的具体含义,因为对于在本章要学的知识而言,不需要知道这些低层的细节。本章中需要掌握的是 MP3 被分成了帧,而每个帧可以有它自己的比特率。

在 MP3 的最末端是音频标记。音频标记包含了 MP3 的曲目、演唱者和专辑的字符串,如图 7.2 所示。它全部是 ASCII 数据,除了音乐流派的类型字段,该字段用一个字节的编码来表示某个特定的音乐流派类型,如表 7.1 所示。

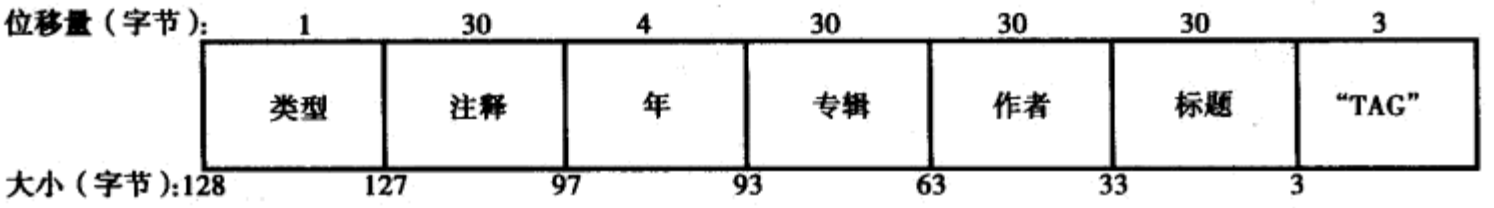


图 7.2 长度为 128 字节的音频标签的格式

表7.1 音乐流派类型编码

编号	音乐流派类型	编号	音乐流派类型	编号	音乐流派类型
0	蓝调音乐	22	Death Metal	44	Space
1	经典摇滚	23	Pranks	45	Meditative
2	乡村音乐	24	音带	46	流行乐器音乐
3	舞曲	25	Euro-Techno	47	摇滚乐器音乐
4	迪斯科	26	环境音乐	48	Ethnic
5	恐怖音乐	27	Trip-Hop	49	Gothic
6	乏味的音乐	28	Vocal	50	Darkwave
7	Hip-Hop	29	爵士乐 & 恐怖音乐	51	Techno-Industrial
8	爵士乐	30	Fusion	52	电子乐
9	金属乐	31	Trance	53	民族流行音乐
10	新世纪音乐	32	古典乐	54	Eurodance
11	稍老的音乐	33	乐器音乐	55	梦幻音乐
12	Other	34	Acid	56	南部摇滚
13	流行音乐	35	剧场乐	57	喜剧音乐
14	R&B	36	游戏音乐	58	Cult
15	敲击乐	37	声音剪辑	59	Gangsta
16	瑞格舞	38	圣经中福音	60	Top 40
17	摇滚	39	噪音	61	基督徒敲击乐
18	Techno	40	Alternative Rock	62	流行乐/恐怖乐
19	Industrial	41	低音	63	Jungle
20	Alternative	42	Soul	64	美国本土音乐
21	斯卡(牙买加流行音乐)	43	Punk	65	歌舞剧

（续表）

编号	音乐流派类型	编号	音乐流派类型	编号	音乐流派类型
66	New Wave	87	Revival	108	Primus
67	梦幻乐	88	Celtic	109	Porn Groove
68	Rave	89	Bluegrass	110	Satire
69	Showtunes	90	Avant-garde	111	Slow Jam
70	Trailer	91	Gothic Rock	112	Club
71	低保真音乐	92	Progressive Rock	113	探戈舞曲
72	Tribal	93	Psychedelic Rock	114	桑巴舞曲
73	Acid Punk	94	Symphonic Rock	115	Folklore
74	Acid Jazz	95	Slow Rock	116	民歌
75	Polka	96	Big Band	117	Power Ballad
76	Retro	97	合唱	118	Rhythmic Soul
77	Musical	98	Easy Listening	119	Freestyle
78	Rock & Roll	99	Acoustic	120	二重奏
79	Hard Rock	100	Humour	121	Punk Rock
80	民间音乐	101	演讲	122	鼓乐独奏
81	Flok-Rock	102	Chanson	123	A Cappella
82	National Folk	103	歌剧	124	Euro-House
83	Swing	104	Chamber Music	125	Dance Hall
84	Fast Fusion	105	奏鸣曲		
85	Bebop	106	交响乐		
86	拉丁音乐	107	Booty Brass		

想要读取音频标记，就请试着从你感兴趣的 MP3 的末尾处的 128 个字节中查找那 3 个 ASCII 字母“TAG”。如果没有找到，那么这个 MP3 就没有音频标记；如果找到了，读取这最

后的 128 个字节并用图 7.2 所示的信息解析它。

下面是实现的代码：

```
bool CMP3AudioTag::Read(std::string filename)
{
  char header[4] = {0};
  char title[31] = {0};
  char artist[31] = {0};
  char album[31] = {0};
  char year[5] = {0};
  char comment[31] = {0};
  char genre = 0;
  int handle = open(filename.c_str(), O_RDONLY);
  lseek(handle, -128L, SEEK_END);
  read(handle, header, 3);
  bool result = false;
  if (! stricmp(header, "TAG")) {
    // 合法的头部……这个 MP3 有一个音频标记!
    read(handle, title, 30);        m_Title = title;
    read(handle, artist, 30);       m_Artist = artist;
    read(handle, album, 30);        m_Album = album;
    read(handle, year, 4);          m_Year = year;
    read(handle, comment, 30);      m_Comment = comment;
    read(handle, &genre, 1);        m_Genre = (MP3_GENRE)genre;
    m_GenreStr = GenreToString(m_Genre);
    result = true;
  }
  close(handle);
  return(result);
}
```

提示

书中列举了庞大的 MP3_GENRE(音乐流派类型),见表 7.1。我创建了一个对象 CMP3AudioTag::GenreToString,该对象将一个 MP3_GENRE 转换成一个 STL 字符串。

这段代码在一个新的类 CMP3AudioTag 中。如果你在 C 语言中使用过文件 I/O 操作，那么这段代码对你来说是简单的。惟一危险的操作是调用 lseek，这个方法将把 128 个字节写回到文件的末尾。在这里的主要操作只是按描述的次序读取数据，并准备就绪。

7.1.3 DirectMusic 用 WAV 格式播放 MP3

现在，我们知道了从来没听过的 MP3 编码为 103 的音乐流派。下面开始学习如何使用 DirectMusic 播放 MP3。

播放 MP3 并不像想象的那么简单。如果尝试加载一个 MP3 文件到加载器使用的段中，将会得到 DMUS_E_UNSUPPORTED_STREAM 错误。你可能会想到用 DirectMusic 加载一个 MP3 文件，但仍然不行。尝试将任何 MP3 文件加载到第 3 章的 WAV 播放程序（Ch3p1_WAVPlayback）中，都将返回这个错误。

有一种方法来让 Ch3p1_WAVPlayback 播放你喜爱的 MP3，并且不用改变代码或者转换格式。这个技巧是：把 MP3 转换成 WAV 文件，这将得到一个大约 30 M 或者 40 M 的文件。虽然可以播放这个 WAV 文件，但别忘了你想要的是 MP3，所以请继续下去。



图 7.3 Windows 录音机

把这个 WAV 文件加载到录音器中（这是个作为 Windows 附属部件的录音程序，如图 7.3所示）。从 File 菜单中选择 Properties，然后单击 Convert Now 按钮。将出现一个对话框（如图 7.4 所示）。

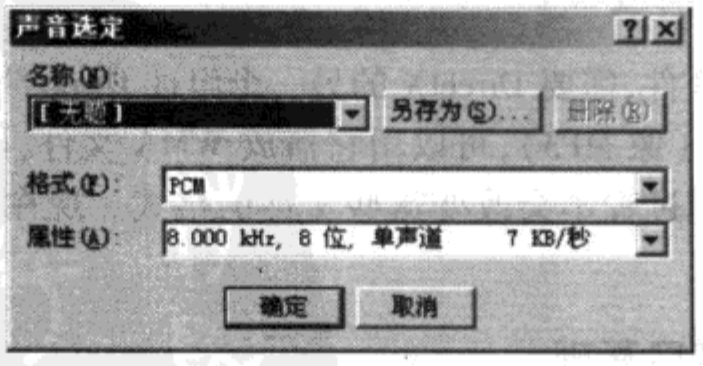


图 7.4 立即转换对话框

从 Format 下拉列表框中选择 MPEG Layer 3。

点击 OK 按钮两次以关闭对话框和属性页,没有什么事情发生。

从 File 菜单选择 Save。现在如果到资源管理器中查看 WAV 文件的大小,就会发现它只有 1 M 或者 2 M(一个 MP3 的大小)。

现在 Ch3p1 肯定可以加载并播放这个 WAV 文件了。

这是怎么回事呢? 首先,请记住 WAV 文件是 RIFF 文件,而 RIFF 可以存储很多不同格式的数据。到现在为止,你都是用 WAV 文件来存储 PCM 格式的数据(非压缩音频),但是 WAV 文件实际上可以存储很多压缩格式,包括 MPEG Layer 3。

提示

转换后的 WAV 文件应该比刚开始的 MP3 要小一些。这是因为转换器只能将 MP3 转换成 56kb/s 的文件,而原始的 MP3 的比特率可能在 128 ~ 192 kb/s 之间。56 kb/s 大大地降低了音质,因此不要期望将它用于实际的游戏中。

如果以二进制的形式打开转换后的 WAV 文件,会发现它仍然有一个 RIFF 头部,只是 PCM 数据替换成了 MP3 数据。这是一个秘密——如果 MP3 有像 WAV 文件的头部,那么 DirectMusic 加载器就可以加载 MP3。RIFF 头部表明:“这个 WAV 使用 MP3 技术进行压缩”,于是加载器使用 MP3 编解码器来解压假冒的 MP3 并把它加载到段中。

正常的 MP3 文件没有头部数据,它们只包含帧的集合和可能在结尾处的音频标记。这使得 DirectMusic 加载器无法识别文件格式,因此它假定是不支持的文件格式并抛出 DMUS_E_UNSUPPORTED_STREAM 错误。但是如果把相同的 MP3 帧打包在 WAV RIFF 头部中,加载器就可以辨识并加载它,不会遇到任何问题。因此,如果想减少操作并且必须通过加载器直接加载文件,而同时又想使用体积小巧的 MP3 文件,那么这是一个最好的折中的方法。

7.1.4 DirectShow 播放 MP3

为播放正常的 MP3 文件,需要 DirectX 的另一个组件 DirectShow。DirectShow 的特长是处理复杂的压缩文件格式(像 MP3),可以用它播放 WMA 文件,以及 AIFF、AU、SND,甚至 WAV 和 MIDI,而且其处理过程不会改变这些文件的格式。现在学习一些有关 DirectShow 工作原理的知识。

(1) DirectShow 接口基础

DirectShow 将滤波器串接在一起工作。一个滤波器本质上针对一种多媒体流进行操作,可以把一个滤波器看成与某种音乐设备或硬件功能相同的一种软件。例如,一个电唱

机的工作是读取唱片上的凹槽并向音频 L/R 输出插孔输出电子脉冲信号;然后将输出连接到一个均衡器的输入(均衡器的惟一工作是调节声波的振幅);最后,把均衡器的输出连接到扩音器的输入(扩音器的工作是把电子脉冲信号输出到扬声器)。

DirectShow 也以相同的方式工作。例如,为了播放 MP3 文件,你可能从 MP3 编解码器开始,MP3 编解码器将解压缩 MP3 并输出音频字节数据;你将编解码器的输出连接到默认 DirectSound 设备的输入。该设备类似于扩音器,它的工作是将音频字节数据输出到扬声器。你所创建的是一个简单的滤波器路径,在 DirectShow 的术语表中,滤波器路径是所有连接在一起的滤波器的集合。

实际上,根本不需要操作滤波器的连接。可以使用 DirectShow 的滤波器路径管理器来自动地为不同类型的源文件创建标准的滤波器路径。只需要告诉滤波器路径管理器播放 MP3,路径管理器就会执行所有连接滤波器的操作。

因此,播放 MP3 所做的几步操作并不复杂。下面讲述它是如何工作的。

(2) MP3 流

要做的第一件事情是创建 3 个接口。IGraphBuilder 接口被用来创建滤波器路径;IMediaControl 接口(与 Windows 的媒体控制接口 API 没有关系)提供播放、暂停和停止一个滤波器路径的操作;IMediaEvent 接口让你可以检测事件(比如,你的 MP3 是否正在播放?)。一些人还创建第四个接口,即 IMediaSeeking,它可以提供虚拟的快速向前和向后的按钮。

在这些接口中,只有路径构造器接口是必须创建的。可以通过查询路径构造器接口得到其他 3 个接口。

下面是创建接口的代码:

```
CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC_SERVER,
    IID_IGraphBuilder, (void **)&pGraphBuilder);
pGraphBuilder -> QueryInterface(IID_IMediaControl,
    (void **)&pMediaControl);
pGraphBuilder -> QueryInterface(IID_IMediaEventEx,
    (void **)&pMediaEventEx);
```

代码创建了一个路径构造器,并且查询另外两个接口。这对于 COM 程序员而言是相当标准的编程方式。IID 在 dshow.h 中定义。

创建了接口以后,可以让路径构造器调用 RenderFile 来创建滤波器路径,如下所示:

```
WCHARwidefilename[MAX_PATH];
DXUtil_ConvertGenericStringToWide(widefilename, filename.c_str());
mp3 -> m_GraphBuilder -> RenderFile(widefilename, NULL);
```

正如 DirectX 中其他组件一样，RenderFile 需要宽字符串形式的路径文件名。第二个参数是保留参数，必须设成 NULL。

有了滤波器路径之后，就可以使用媒体控制接口来控制虚拟播放按钮：

```
ThrowIfFailed( pMediaControl -> Run( ), "Run failed." );
```

此外还有用于停止(Stop)和暂停(Pause)的媒体控制接口方法。

但是 Stop 不会自动移到 MP3 的开头处。为了能够移到开头，需要使用 IMediaSeeking 接口。

当 MP3 播放完以后，可以释放掉(Release)所有的接口。按照良好的 COM 编程策略，最好按创建时相反的次序来释放它们：先是 media control，media seek(如果用到了)，media event，最后是 IGraphBuilder。

(3) DirectShowd 的消息

当在播放、停止和管理滤波器路径时，DirectShow 发送的消息被称为事件。IMediaEvent 接口用于接收这些消息，如下所示：

```
long eventcode, param1, param2;
while (m_mediaEventEx - > GetEvent(
  &eventcode, &param1, &param2, 0) == S_OK) {
  if (eventcode == EC_COMPLETE) {
    // MP3 播放完
  }
  m_MediaEventEx -> FreeEventParams(eventcode, param1, param2);
}
```

这段代码与在上一章的 DirectMusic 通知中所做的事情很类似。调用 GetEvent 函数本质上是把第一个事件从内部队列中取出来，然后放到所给出的存储空间里。如果没有更多的消息，GetEvent 将不返回 S_OK，循环结束。当事件处理完毕，必须调用 FreeEventParams 来清除 DirectShow 分配的内存空间。

下面是如何在音频引擎中加入对 MP3 的支持。

(4)让音频引擎支持 MP3

对于初学者，需要连接 strmiids. lib，它包含了将要使用的接口的 ID。

接着是一些新类。我创建了 CDirectShowMusic 类，它代表你想通过 DirectMusic 播放的任意类型的音频文件。CDirectShowMusic 类从顶层的通用音频对象 CSound 继承而来。

从 CDirectShowMusic 类继承得到一个子类 CMP3，它用来添加任何与 MP3 相关的操作

(比如管理它们的音频标记)。为了便于音频标记的管理,我又创建了一个新类CMP3AudioTag,在前两节已经见过这个类。

现在,CMP3 实际上是一个空的子类。它只包含一个 CMP3AudioTag。所有有趣的代码都在 CDirectShowMusic 中,这个类实现了纯虚函数 Play、Stop 和 IsPlaying(在 CSound 中已经定义)。

要让音频引擎增加对 MP3 的支持,最困难的是处理事件。没有简单的方法来查询滤波器路径是否正在播放,因此音频引擎需要一个实现这种功能的方法,使 IsPlaying 函数能够可靠地工作。

可能想到的办法是在构造器中创建一个布尔型变量并设置成假值。当 Play 被调用时,将该变量设置成真值;当 Stop 被调用时,将该变量设置成假值。但是,这种方法不能处理 MP3 文件全部播放完后自己停止的情况。为了捕获这种情况需要检测 DirectShow 事件。

于是,可以在音频管理器的 DispatchNotificationMessages 函数中加入了新的代码(该方法在上一章中已讲述)。DispatchNotificationMessages 现在调用 CDirectShowMusic 的一个静态函数 DispatchDirectShowNotifications。DispatchDirectShowNotifications 遍历当前所有被实例化的 CDirectShowMusic 类,并且调用非静态函数 ProcessMyDirectShowNotifications。该方法像前一节中所描述的那样工作:它分别调用 GetEvent 函数,直到队列中没有事件为止。如果它遇到 EC_COMPLETE 事件,就把布尔变量 m_Isplaying 设置为假值。这就确保了 m_Isplaying 始终都是正确的。

最后,我在 CAudioManager 中加入了一个新函数 LoadMP3,它加载 MP3 到一个新的 CMP3 对象中,并且返回它的 CSoundPtr。请仔细学习这些代码,确信没有不清楚的地方。

7.2 播放 WMA

WMA(Windows Media Audio)是微软提出的一种声频压缩格式,它宣称是比 MP3 更好的压缩文件格式。微软声称在相同的存储空间里,WMA 文件的容量是 MP3 文件的两倍,而一个 64 kp/s 的 WMA 文件的音质与 128 kp/s 的 MP3 的音质相当。

读者必须自己判断它们对不对。Internet 上关于哪种文件格式最好的争论以及人们按自己的规则对音频文件格式的排序,其实都是为了寻找真正的“最好的”。在本书光盘中包含了一些相关研究的链接。

如果使用 WMA,那么会发现操作是非常简单的。只需要简单地使用 DirectShow 的 MP3 播放代码就可以了。它也可以播放 WMA 文件。

这里只是顺便提一下 WMA。需要认真学习的知识还在后面。

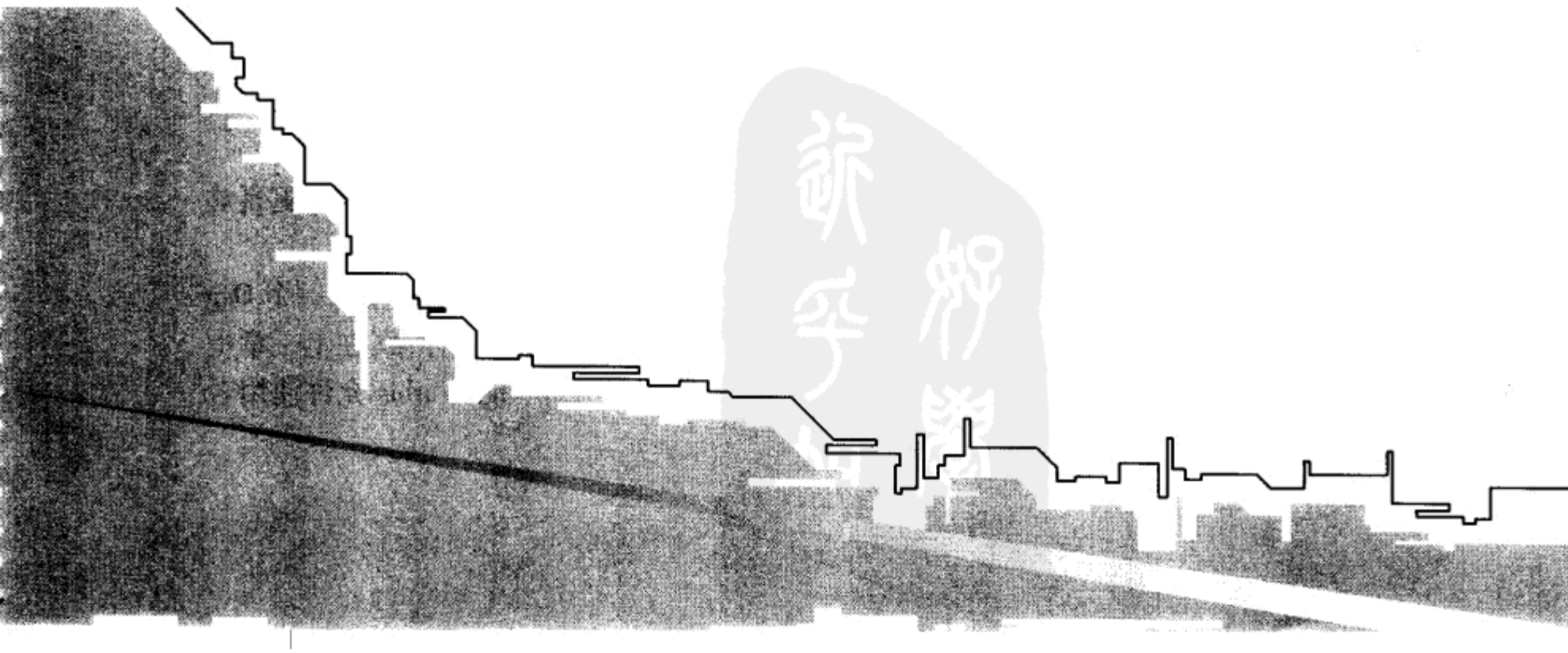
7.3 本章小结

本章简单介绍了 MP3 和 WMA，介绍了如何使用 DirectShow 播放 MP3 文件格式，学习了一点关于 MP3 文件格式的知识，以及让 MP3 表现成 WAV 格式的一个技巧。

下一章将学习另一种用于替代 MP3 格式的音频文件格式 Ogg Vorbis。

第 8 章

播放 OGG VORBIS



两种主要的压缩音频格式，我们已经学习了其中的一种。本章将进一步扩展音频引擎，以支持 Ogg Vorbis 编码格式。Ogg Vorbis 是由 Xiphophorus 组织(xiph. org)创建的一种开放源代码的编码格式。因为它的代码是开放的，并且很快具备了取代 MP3 的能力，所以它在 Linux 上非常流行。

使 Ogg Vorbis 出众的原因是它是开放源代码。它本身的规范是免费的，任何人可以编写代码来读取或者修改 OGG 文件，而不用为许可证付费。此外，在 BSD 许可下 Ogg Vorbis 库是可用的，这意味着可以在自己的应用程序中全部或者部分使用这些类库。

提示

如果你在游戏中使用了 Ogg Vorbis，我恳请你为 Ogg Vorbis 组织做些事情作为回报。开放源软件的根本思想是一个人改进了一件事情，然后以开放源的形式发布它，让其他人可以进一步的改进它。请向 Xiph. org 捐点款，让他们可以持续发布好的代码，并且/或者捐献一些智力成果，以推动编解码器的发展。

音频剪辑

音频剪辑 8.1 和 8.2 对比了一个 WAV 文件和相应的 Vorbis 编码的文件。

只有在自己进行比较的基础上，才能客观地评价 Ogg Vorbis 和 MP3、WMA 等格式，并决定哪一种格式最适合自己。我个人认为 Ogg Vorbis 的音质比 MP3 好，但有人会质疑我的欣赏水平。分割 WAV 文件技术上等同于 MP3 和 Ogg Vorbis 格式，看你更喜欢哪一个。

下面介绍的内容不能用于 MP3 文件格式，除非支付了相应的费用。本章直接将一个 Ogg Vorbis 解码器嵌入到音频引擎中。MP3 需要使用者支付许可证费用才能这样做(在上一章中回避了这个问题，因为微软的 DirectShow 做了解码工作)。另一方面，Ogg Vorbis 允许使用者解码自己的格式，也不需要付费。

8.1 Ogg Vorbis 类库结构

Ogg Vorbis API 有一套不同的类库，每一个都被设计用于一个特定的目的。Ogg Vorbis API 既可以静态链接，也可以使用 DLL。静态类库是自包含的，不需要额外的 DLL 文件，因此对于大多数的游戏都可以使用静态类库，以避免 DLL 冲突。当然也可以使用依存(Stub)类库，它需要操作某些 DLL，可以在 bin 文件下找到这些 DLL。

Ogg Vorbis API 被分成了 4 个不同的类库：Ogg 类库、Vorbis 类库、Vorbisenc 类库和 Vorbisfile 类库。

8.1.1 Ogg 类库和 Vorbis 类库

这两个类库包含了操作 Ogg Vorbis 文件所需要的基本函数。

Ogg 类库包含了处理 Ogg 文件的通用例程。实际上，Ogg 类库不知道(也不关心)你所管理的是音频数据还是视频数据。它只负责创建基本框架。至于 Vorbis 类库，则包含使用 Vorbis 解码器解码流数据时所必需的基础函数。

Ogg Vorbis：重要的编解码器

对这个奇怪的名字有一些注解：Xiph. org 为压缩音频和视频创建了一个框架。这个框架被称为 Ogg。实际的音频编解码器被称为 Vorbis。RIFF 文件可以包含 MP3 编码的数据，类似地，Ogg 文件也可以包含 Vorbis 编码的数据。当前，Vorbis 编解码器是 Ogg 框架中所使用的惟一的编解码器，但是 Xiph. org 计划发布其他编解码器以用于广泛的应用程序。这些发布的编解码器也将使用 Ogg 框架，因此通常认为 Ogg 位于 Vorbis 之上。

8.1.2 Vorbisenc 类库

这个类库被设计用来帮助编码 Vorbis 文件。在这儿，可以找到加载和卸载 Vorbis 编码器的函数。如果你正在创建一个应用程序来把 CD 音频转换成 Ogg Vorbis 文件，这将是你自己的类库，但游戏通常不需要使用它。

8.1.3 Vorbisfile 类库

这是一个有用的类库，用于处理 Ogg Vorbis 文件。这个类库使用 Ogg 类库和 Vorbis 类库，并且增加了一些代码，用来创建一个更加简单的接口，特别是在解码文件时。这个类库最适合于游戏，因此，它是本章学的最多的类库。

8.1.4 获得帮助

在开始学习前,还要提及一件事情,那就是如何得到 Ogg Vorbis 类库的帮助文档。在安装路径下的 doc 目录里有一个叫做 index. html 的 HTML 文件,它是文档的根。如果想得到特定类库的帮助文档,可以直接到 doc 目录的子目录下(ogg,vorbis,vorbisenc 或 vorbisfile),打开相应目录里的 index. html 文件,获得相应的帮助信息。

8.2 安装 Vorbis

安装类库:在光盘里 Goodies 目录下可以找到几个 Ogg Vorbis ZIP 文件。在光盘里包含了所有你可能感兴趣的文件,但本章只需安装两个文件,它们是 libvorbis-1.0. zip 和OggVorbis-win32sdk-1.0. zip。下面将讲述如何安装这些文件。

先把两个 zip 文件中的所有文件拷到一个目录树中,分别解压两个文件到独立的目录里(要保持 zip 文件中的目录结构),然后将两个目录中的内容拷贝到一个主目录里(覆盖文件是安全的)。

无论哪种方式,最终得到的目录结构包括 lib、Win32 example 和其他目录分支。特别注意要确保 include 目录中有 ogg 目录和 vorbis 目录。

①打开命令行控制台(在 Windows NT,Windows 2000 或 Windows XP 上,从“开始”→“运行”,键入**cmd**;或者在 Windows 95,Windows 98 和 Windows Me 上,选择**command**)。

②需要设置一个环境变量,该变量叫做 SRCROOT。为了正确地安装类库,SRCROOT 必须指向所解压文件的顶层目录。比如说,如果拷贝两个目录到一个单一的目录,该目录名是 C:\libraries\oggvorbis,那么在命令行控制台中,应该输入**set SRCROO = C:\libraries\oggvorbis**。作者强烈建议安装路径名不要有空格,以避免在引用该路径时遇到麻烦。

③输入**cd %SRCROOT% \win32**。如果正确设置了 SRCROOT 环境变量,应该转到 win32 目录下,C:\libraries\oggvorbis\win32。

④在这个目录里是批处理文件,这些批处理文件创建各种不同的 Ogg 类库和 Vorbis 类库。如果愿意,你也可以运行每一个批处理文件,但是现在只需要运行 build_vorbis_static. bat、build_vorbis_static_debug. bat、build_vorbisfile_static. bat 和 build_vorbisfile_static_debug. bat。这几个批处理文件将分别创建 vorbis_static. lib、vorbis_static_d. lib、vorbisfile_static. lib 和 vorbisfile_static_d. lib。请注意,在编译这几个文件的时候会得到一些警告信息。

笔者在创建 Ch8p1_OggVorbisPlayback 时就是依照上述步骤的。在第 8 章的源代码目录里,可以找到一个叫做 libvorbis-1.0 的目录,它包含了编译好的类库。

接下来的一步是创建使用 Ogg Vorbis 类库的 DevStudio 项目。把 Ogg Vorbis 添加到一个

项目中所要做的事情步骤如下：

①将包含 Ogg Vorbis 的路径添加到项目中。其操作是：从 Project 菜单选择 Settings 项，然后转到 C/C ++ 选项。从 Category 下拉菜单中选择 Preprocessor。在 Additional Include Directories 编辑框里，输入从项目到包含 Ogg Vorbis 目录的相对路径。比如，对于本章的示例程序，输入.. \.. \libvorbis-1.0\include 。

②链接 Ogg Vorbis 类库。其操作是：转到设置对话框的 Link 选项，选择 Input。在标示为 Object/library modules 的编辑框里，添加要使用的 Ogg Vorbis 类库。对于本章的示例程序，添加了 vorbisfile_static. lib，ogg_static. lib 和 vorbis_static. lib。

③将 Ogg Vorbis 库目录添加到类库路径中。在相同地方，Object/library modules 编辑框的下面是 Additional Library Path 编辑框。在这里添加 Ogg Vorbis 库目录的相对路径。对于 Ch8p1_OggVorbisPlayback 示例程序而言，就是.. \.. \libvorbis-1.0\lib 。可用逗号分隔多个不同的路径。

④确保代码中包含了正确的 Ogg Vorbis 头文件以及在路径中使用 vorbis/。比如，如果想包含 vorbisfile. h，应该输入#include <vorbis/vorbisfile. h>。预处理器会添加在第 1 步中加入的相对路径，得到.. \.. \libvorbis-1.0\include\vorbis\vorbisfile. h。

8.3 使用 Vorbis

读到这里，读者应该已经成功地解决了在处理第三方 API 时遇到的困难：安装和配置类库。现在来看一些 Ogg Vorbis 代码。

8.3.1 使用 Vorbisfile API

Vorbisfile API 是最简单的 Ogg Vorbis API。它使你可以方便地读取 Ogg Vorbis 文件并将它们转换成非压缩的 PCM 流。图 8.1 描述了如何使用这个 API。也可以使用其他的 API(查看 Ogg Vorbis 帮助文档)，但在本章将只使用 Vorbisfile API。

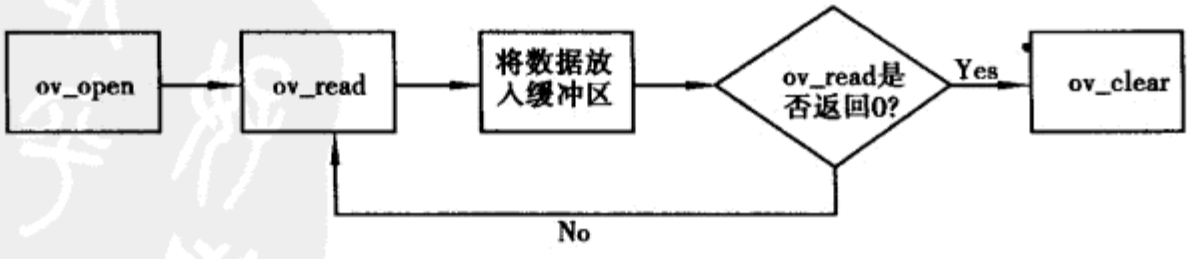


图 8.1 使用 vorbisfile API 的流程图

这个 API 按照将读取文件的过程分解来组织结构。为了操作一个文件,要打开它,读取它,然后在操作完以后关闭它。类似的,在对 Ogg Vorbis 文件进行操作时,必须调用 ov_open API 来打开它,调用时需要传递的参数有:用 fopen 打开的文件句柄,以及需要被赋值的 Ogg Vorbis_File 结构的地址。

打开 Ogg Vorbis 文件以后,可以做一些不同的操作。可以调用 ov_info API 函数来得到它的一些信息。调用时需要传递的参数有:指向 OggVorbis_File 结构的指针(就是从 ov_open 返回的指针),以及需要被赋值的 vorbis_info 结构地址。vorbis_info 结构在 vorbis/codec. h 中声明,它包含的信息有文件的版本、声道数量、采样率和可能的比特率(这取决于文件的比特率是否可以变化)。

其他有用的 API 函数有 ov_pcm_total 和 ov_comment。ov_pcm_total 可以说明整个样本的数量,而 ov_comment 则显示用户在文件中的注释(演唱者、专辑、曲目等)。

最有用的 API 函数是 ov_read。它将一个文件解压成 PCM 流。ov_read 的输入参数有指向 OggVorbis_File 结构的指针、一个字节缓冲区、缓冲区的长度(包括一些参数用以表明你想用 big-endian 还是 little-endian 输出)、平台的字的长度、以及想要有符号数据还是无符号数据。

ov_read 的最后一个参数是一个指向整数的指针,这个整数指示当前的逻辑比特流。对于简单的解码,只需要把整数设成 0,并将这个整数的地址传给 ov_read。这让你不用操心逻辑比特流;如果读者有更多兴趣的话,可以查看 Ogg Vorbis 文档。

ov_read 可能不会填充完你所给的字节缓冲区,因此检查返回值很重要,这个值表明得到了多少个字节的数据。如果返回值为 0,则表示已在 Ogg Vorbis 流的结尾处了。

读取完文件以后,用 ov_clear API 函数关闭它。请注意这个操作也会关闭 FILE 指针;因此没有必要调用 fclose 方法。

8.3.2 将 Ogg Vorbis 集成到音频引擎中

现在,类库已经安装好了,并且也知道了 API 是如何工作的,可以开始编写一些代码了。在本节,将学习如何加载 Ogg Vorbis 文件到 DirectMusic 段中。需要做的事情是把 Ogg Vorbis 文件解压成 PCM 流(非压缩流),把 PCM 流封装在 WAV 文件头部,然后传给 DirectMusic 的加载器加载到段中。

读者可能会遇到一些困惑。不过接下来一个 Ogg Vorbis 示例程序(vorbisfile_example. c)将演示如何使用 vorbisfile 类库来将 Ogg Vorbis 文件转换成 PCM 流。在第 4 章中“加载 WAV 文件”一节我们已经编写了 CWAVFile 类,它可以从非压缩的数据块中创建 WAV 文件。

我在 CAudioManager 中加入了新函数 LoadOggVorbis。下面是该函数源代码:

```
CSoundPtr CAudioManager::LoadOggVorbis(std::string filename)
{
```

```
std::vector<unsigned char> pcmdata;
// 这段代码基于 vorbisfile_example.c,包含在 ogg 的发布中
char pcmout[4096]; // ogg 解码缓冲区
OggVorbis_File vf;
int eof=0;
int current_section=0;
FILE *f = fopen(filename.c_str(), "rb");
if (f == NULL) { Throw("CAudioManager::LoadOggVorbis: could not open file."); }
if(ov_open(f, &vf, NULL, 0) < 0) {
  Throw("Input does not appear to be an Ogg bitstream.");
}

/* 抛出注释以及几行关于正在解码的比特流信息 */
vorbis_info *vi=ov_info(&vf, -1);
// 将 pcmdata vector 的大小变成 ogg 文件的大小
// 每个样本每个声道两个字节
int samples = ov_pcm_total(&vf, -1);
int channels = vi->channels;
int samplerate = vi->rate;
if (samples > 0) {
  pcmdata.reserve(channels*2*samples);
}
while(!eof){
  long ret=ov_read(&vf,pcmout,sizeof(pcmout),0,2,1,&current_section);
  if (ret == 0) {
    /* EOF */
    eof=1;
  } else if (ret < 0) {
    /*流中出现错误。在我们(应用程序)这种情况下,不是问题,只是报告。现
      在,我们不用报告。*/
  } else {
    /* 我们没有处理采样率的变化等等,但你以后必须做。*/
    // 从 ogg 缓冲区中传递数据给我们。
```

```
            for (int q=0; q < ret; q++) {
                pcmdata.push_back(pcmout[q]);
            }
        }
    }
    /* 清除 */
    ov_clear(&vf);

    // 现在我们在内存中有了 PCM 数据,把它转换成 WAV 文件
    // 并传给 DirectMusic 加载到段中。
    CWAVFile wavfile;
    wavfile.m_AudioFormat = WAVE_FORMAT_PCM;
    wavfile.m_BitsPerSample = 16;
    // 16 位 / 8 位 每字节 * 声道数量
    wavfile.m_BlockAlign = 2 * channels;
    wavfile.m_ByteRate = samplerate * channels * 2;
    wavfile.m_NumberOfChannels = channels;
    wavfile.m_SampleRate = samplerate;
    wavfile.SetData(pcmdata.begin(), pcmdata.size());
    return(LoadSound(wavfile));
}
```

大部分的源代码都是从 vorbisfile_example.c 剪切拷贝过来的。vorbisfile_example.c 从标准输入(stdin)中读取 Ogg Vorbis 流,并输出非压缩的 PCM 流到标准输出(stdout);我做了一些修改,使它可以从一个 FILE 指针中读取 Ogg Vorbis 流,并把它写到一个无符号字符串中。

请注意这段代码知道输出的 wave 是 16 位的,因为它通过传递 2 作为第 5 个参数来指定 16 位字长给 ov_read 函数。如果想要一个 8 位的 WAV 文件,可以传递 1。

在 PCM 流存入 pcmdata 数组后,代码将它传给一个 CWAVFile 实例(wavfile),包括其格式的细节信息(声道数量、采样率等)。然后,调用在第 4 章中介绍的 LoadSound 函数来读取 WAV 文件类并创建一个 DirectMusic 段。

8.4 本章小结

本章学习了如何使用开放的 Ogg Vorbis 编解码器，并用其作为 MP3 和 WMA 的替代方法。然而，解码远远不止 Ogg 和 Vorbis。我建议读者尝试其他的 Vorbis 类库，并阅读 Vorbis 的技术规范。

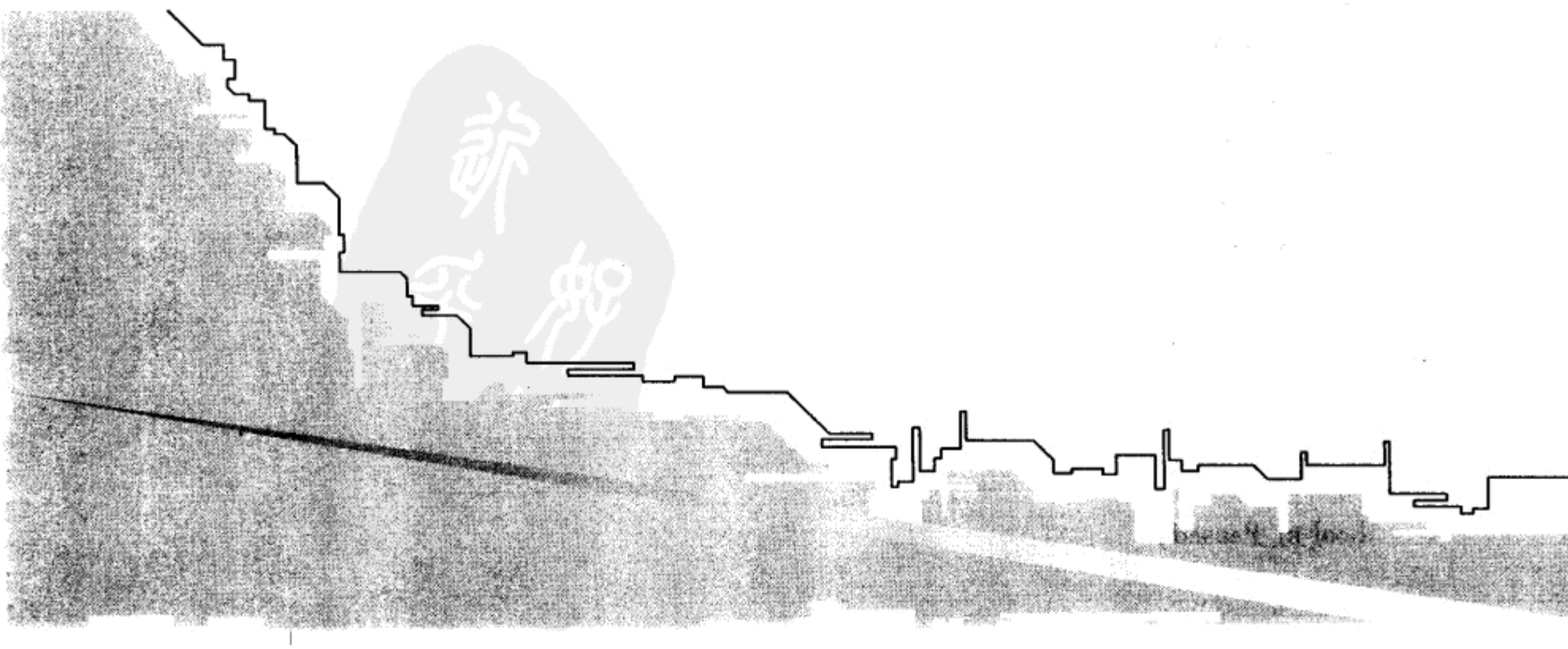
本章主要涵盖了音频压缩文件格式。在下一章，将学习如何读取和播放音轨音乐格式，它主要用在游戏演示场景中，你会发现它在游戏编程中很有用。

PDG

PDG

第9章

播放 CD 音频



如果游戏是基于光盘的，则会有既简单又多样化的途径来实现游戏音乐。除非想压缩光盘（且没有空间存储非压缩音频），或者需要音乐在运行时动态变化，本书建议使用 CD 音频作为游戏音乐。

本章将集中讲解 CCDPlayer 类的构建。下面是这个类的声明，取自于 Ch9p1_CDPlayer 目录里的 CDPlayer.h 文件：

```
class CCDPlayer
{
  public:
  CCDPlayer();
  virtual ~CCDPlayer();
  void Init(HWND hwnd, char driveletter = 0);
  void UnInit();
  void Play(int track);
  bool IsPlaying();
  bool IsPaused();
  void Stop();
  void Pause();
  void Resume();
  void OpenCDDoor();
  void CloseCDDoor();
  bool IsCDInDrive();
  CTrackMinSecFrame GetCurrentPos();
  int GetTrackLength(int track);
  int GetNumTracks();
  static std::vector<CCDDeviceInfo> GetAvailableDevices();
  protected:
  bool m_InitGood;
  bool m_Paused;
```

```
    CTrackMinSecFrame m_PausedPos;
    static void HandleError(MCIERROR err);
    HWND m_hWnd;
    DWORD m_DeviceID;
};
```

正如所看到的,这段代码是一个没有附加功能的 CD 播放器(没有考虑在音轨里搜索或检测 CD 驱动器),它只具有简单的功能(游戏所需要的功能,没有额外的功能)。

这个类依赖于另外两个类:CTrackMinSecFrame 和 CCDDeviceInfo。这两个类实际上只是公共变量的集合。CTrackMinSecFrame 包含 4 个整数:m_Track、m_Minute、m_Second 和m_Frame,它们共同定义了 CD 中的一个时间点。CCDDeviceInfo 包含了一些描述,给定 CD 音频驱动器的字符串。在这儿保存了驱动器的盘符(m_DriveLetter)、驱动器的简短描述(m_Description)、UPC 代码(m_CDProductCode)和驱动器中光盘可能存在的 ID(m_CDID)。所有这些信息都来源于 Windows。

现在,用 MCI 来充实这个类声明。

9.1 用 MCI 播放 CD 音频

第 3 章已经介绍了 Windows 的媒体控制接口 MCI(Media Control Interface),它被设计用于多媒体设备,包括支持 CD 播放。MCI 是使用 CD 最简单的方式,也有更低层操作的方法,除非需要编写音频提取软件,否则不需要使用它们。

图 9.1 描述了使用 MCI 的过程:查看哪些设备可用,选择其中一个并打开它,对它发出操作命令,在操作完后关闭它。可以通过 API 函数进行交互,或可以格式化一个文本命令字符串并发送。比如,命令字符串"play thissound from 0 to 2 000 notify"就是告诉系统从开始处到位置 2 000 播放"thissound",并在播放完后通知你。

MCI 类库是一个功能强大的工具,涵盖了大部分游戏使用 CD 播放器时所涉及的操作。如果想得到整个 API 的信息,可查看在 Platform SDK\Multimedia Audio\MCI 下的 MSDN 帮助文档。

MCI 是一个双接口 API,可以使用自由格式的文本命令(命令字符串接口)或者使用消息 ID 和结构集合(命令消息接口)来与之交互。本书忽略自由格式字符串接口,选择使用命令消息接口。

消息接口的主要思想体现于 mciSendCommand 调用。这个函数带的参数是命令消息 ID(如 MCI_PLAY),以及与消息相关的结构(如 MCI_PLAY_PARMS)。当把结构的地址传给函数时,需要把地址转换成 DWORD,其代码段如下:

```
MCI_OPEN_PARMS micPlayParms;
ZeroMemory(&micPlayParms, sizeof(MCI_OPEN_PARMS));
micPlayParms.dwFrom = MCI_MAKE_TMSF(bTrack, 0, 0, 0);
micPlayParms.dwTo = MCI_MAKE_TMSF(bTrack + 1, 0, 0, 0);
micPlayParms.dwCallback = (DWORD) hWndNotify;
MCIERROR e = mciSendCommand(wDeviceID, MCI_PLAY,
  MCI_FROM | MCI_TO | MCI_NOTIFY, (DWORD)(LPVOID) &mciPlayParms);
```

这段典型的代码取自 CD 播放器类的 Play 函数。你可以看到代码设置了 MCI_PLAY_PARAMS 结构的各个变量,这个结构是 MCI_PLAY 命令所需要的。代码通过 dwFrom 和 dwTo 变量指定播放范围,返回一个 Window 句柄,然后调用 mciSendCommand 来发送命令。mciSendCommand 的第二个参数是消息 ID,第三个参数是有关消息标记的集合。

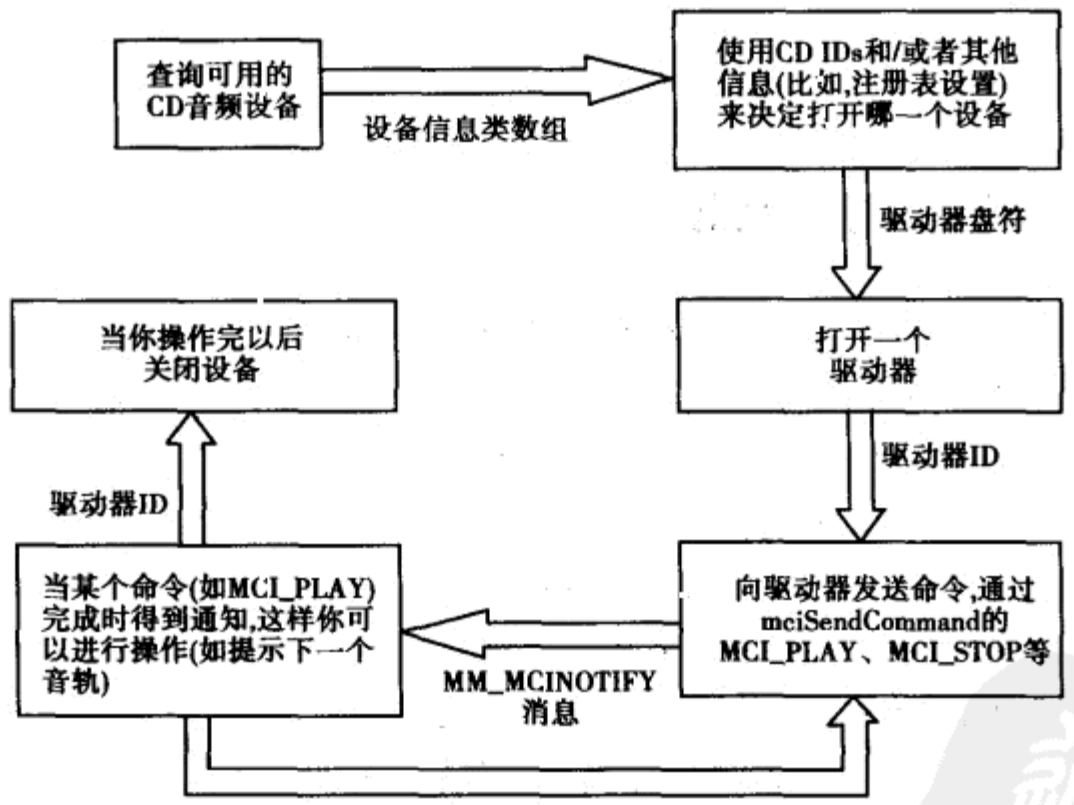


图 9.1　使用媒体控制接口(MCI)的过程

不用考虑具体细节,只需了解参数模式,因为它在整个 CD 播放器类中都将使用。如果 mciSendCommand 操作成功,返回 0;反之,则返回一个 MCI 错误代码,可以用 mciGetErrorString 函数将错误代码解码成文本字符串。

9.1.1 枚举、打开和关闭设备

程序逻辑的首要事情是判断游戏玩家的硬件种类。过去这件事比较简单,因为当时大部分的计算机都只有0个或1个设备。由于CD驱动器和USB的增长,使这件事变得更复杂了。从理论讲,如今一个用户可以有许多外部设备能够播放CD。游戏光盘只是其中一个设备,那么该如何判断是哪一个设备呢?

需要指出一点,通常"他们"让你做事的方式并不需要复杂的操作。如果正在阅读MSDN文档中关于MCI API的内容,那么你会很快发现枚举设备的方法只要发送一次带有某些标记的MCI_SYSINFO命令,便可以获得MCI设备的数量,然后,再将带不同标记的命令多次发送给每个设备。

通常还可以采用更简单的方法。Windows自动给每一个CD音频外设(CD播放器、DVD播放器、CD刻录机等)分配一个从D到Z之间的盘符(盘符A和B通常用于软驱,盘符C通常用于硬盘驱动器)。因此,可以从字母D到Z进行遍历,查看是否有一个CD音频设备连接到了这个盘符。如果有,可以与该驱动器建立通信,判断驱动器里的光盘是什么。这种方式比MCI枚举方法更简单,并且它可以告诉你游戏光盘所在的驱动器盘符。

下面是CCDPlayer::GetAvailableDevices的方法,它查看系统中的所有驱动器,并返回一个设备信息类的数组——一个设备信息类对应它所找到的一个CD音频驱动器:

```
vector < CCDDeviceInfo >  CCDPlayer::GetAvailableDevices( )
{
  DWORD retvalue = 0;
  MCIERROR e;
  vector < CCDDeviceInfo >  devices;
  // 遍历整个驱动器盘符的字母表
  for (char drive  = 'C'; drive  <= 'Z'; drive ++ ) {
    char buf[256];
    char driveletter[3]  =  "C:";

    // 查看我们是否能以CD音频驱动器打开该设备
    MCI_OPEN_PARMS openparms;
    ZeroMemory(&openparms, sizeof(MCI_OPEN_PARMS));

    openparms. lpstrDeviceType  =  (LPCSTR) MCI_DEVTYPE_CD_AUDIO;
    driveletter[0]  =  drive;
```

```
openparms.lpstrElementName = driveletter;
e = mciSendCommand(NULL, MCI_OPEN,
  MCI_OPEN_TYPE | MCI_OPEN_TYPE_ID | MCI_OPEN_ELEMENT,
  (DWORD)(LPVOID)&openparms);
if (!e) {
  // 成功! 查询这个 CD 音频驱动器,并把它加到我们的数组中。
  CCDDeviceInfo devinfo;
  devinfo.m_DriveLetter = drive;
  int deviceid = openparms.wDeviceID;
  //从列表中删除,以节省版面
  // 查询驱动器的描述、驱动器里面的光盘的产品代码和 ID
  //关闭设备
  MCI_GENERIC_PARMS genericparms;
  ZeroMemory(&genericparms, sizeof(MCI_GENERIC_PARMS));
  mciSendCommand(deviceid, MCI_CLOSE, MCI_WAIT,
    (DWORD)(LPVOID)&genericparms);

    // 把它加入到数组中
    devices.push_back(devinfo);
  }
} // 下一个盘符

  return(devices);
}
```

作者对 GetAvailableDevices 函数进行了删减,以便让读者更容易了解整体的结构。

实质上,这段代码遍历了字母表,尝试以 CD 音频驱动器打开每一个驱动器盘符。最重要的是 MCI_OPEN_ELEMENT,它触发 MCI 去查看 MCI_OPEN_PARMS 结构的 lpstrElement-Name 变量。这里,lpstrElementName 实际上是驱动器的盘符。

如果所给的驱动器盘符是 CD 音频设备,代码将接收到操作成功返回的代码 0,它来自于 mciSendCommand 所发送的 MCI_OPEN 消息。这就告诉代码该盘符所指向的驱动器能够播放音频光盘。于是代码发送一些 MCI_INFO 消息,查询驱动器的描述、光盘产品代码和 ID。在上面省略了这段代码,它相当简单。

提示

一个光盘的通用产品代码(UPC)是生产商刻在光盘上的,并且被认为是惟一的。这有点像杂货店里商品的条纹码。比如,可以用光盘的 UPC 在 Internet 上来查询 CDDB(光盘数据库),得到光盘的标题、作者和音轨名。但要注意:有些旧的光盘没有产品代码。和光盘产品代码不同,CDID 代码是一个伪标示符,由 Windows 根据音轨的长度和光盘的其他属性计算而来。Windows 可以为任何一张光盘生成 ID,但是有可能(尽管不太可能)两张不同的光盘有相同的 CDID。

代码查询完所有设备后,将发送 MCI_CLOSE 命令关闭设备。请注意 MCI_WAIT 标记,默认 MCI 命令是异步工作的,通过 MCI_WAIT 标记告诉 MCI 在设备处理完消息前不要返回。代码等待关闭命令完成,然后再加入设备信息类到数组,并转到下一个驱动器盘符,这样做是安全的。

这个静态函数返回一个 CCDDeviceInfo 类的数组给调用者。当调用者得到合法的驱动器盘符列表以后,从中选择一个,然后调用 CCDPlayer 的 Init 初始化函数。这里没有给出 Init 源代码,因为它很简单。Init 打开与所给盘符相对应的驱动器,并把它的设备 ID 放入 m_DeviceID 中。这个设备 ID 有点像文件的句柄;代码在打开消息的响应中接收到设备 ID,并把它作为更多有用消息的目标,像 MCI_PLAY。这就是如何在装有多个 CD 音频驱动器的系统中获得正确光盘的方法。

当驱动器被打开后,Init 函数发出 MCI_SET 消息,设置时间格式 TMSF,TMSF 包含了精确指定音频光盘上某个确切位置的 4 个标示符:音轨、分、秒、帧。当然也有其他时间格式(比如,毫秒),但是如果主要是播放音轨的话,TMSF 将是最好的选择,读者们在学习播放音轨时就会理解其原因。

最后,Init 函数设置 m_InitGood 标记为真值。这个标记是“批准标签”,表示 CD 播放器准备好了;CCDPlayer 的其他函数都会在进行操作前检查这个标记。

用完 CD 播放器类以后,调用它的 UnInit 函数,发送 MCI_CLOSE 消息以关闭设备。

9.1.2 MCI 时间格式

先讲 CD 时间格式。我写了一个辅助类 CTrackMinSecFrame,它包含了用于音轨、分、秒和帧的整数。而 MCI 函数则把这 4 个值封装在一个单一的 DWORD 中,一个字节对应一个值。MCI 提供了宏来存放和提取 TMSF DWORD,可以使用这些宏来创建函数,以便在 CTrackMinSecFrame 和 DWORD 之间转换:

```
class CTrackMinSecFrame
```

```
{
  public:
  //删减
  CTrackMinSecFrame &operator = (DWORD dw) {
    m_Track = MCI_TMSF_TRACK(dw);
    m_Minute = MCI_TMSF_MINUTE(dw);
    m_Second = MCI_TMSF_SECOND(dw);
    m_Frame = MCI_TMSF_FRAME(dw);
    return( *this);
  }
  operator DWORD() { return(MCI_MAKE_TMSF(m_Track, m_Minute,
    m_Second, m_Frame)); }
};
```

这里,有两个操作符被重载。第一个是 operator =,它使用了 4 个 MCI 宏来分别提取音轨、分、秒和帧字节,并把它们放入新的整数中。第二个是函数 operator DWORD,它使用 MCI_MAKE_TMSF 把这 4 个值放回去。

这些操作符使得我们可以把 CTrackMinSecFrame 看成是 DWORD。C ++ 的经验是:当习惯类型自动转换时,操作符重载会使事情变得简单。

9.1.3 播放音轨

不能简单告诉 CD 播放器“播放第 8 音轨”,而必须告诉它开始时间和结束时间。由于使用了 TMSF 格式,所以操作起来相当简单,见如下代码:

```
void CCDPlayer::Play(int track)
{
  if (!m_InitGood) return;
  m_Paused = false;
  MCIERROR e;
  MCI_PLAY_PARMS mciPlayParms;
  mciPlayParms.dwFrom = 0L;
  mciPlayParms.dwTo = 0L;
  mciPlayParms.dwFrom = MCI_MAKE_TMSF(track, 0, 0, 0);
  mciPlayParms.dwTo = MCI_MAKE_TMSF(track + 1, 0, 0, 0);
  mciPlayParms.dwCallback = (DWORD)m_hWnd;
```

```
    e = mciSendCommand(m_DeviceID, MCI_PLAY,
      MCI_FROM | MCI_TO | MCI_NOTIFY,
      (DWORD)(LPVOID) &mciPlayParms);
    if (e) HandleError(e);
}
```

通过使用TMSF,代码可以方便地指定播放范围,从所指定的音轨的开始处开始(track,0,0,0),直到下一个音轨结束,但不包含下一个音轨(track +1,0,0,0)。TMSF为控制播放提供了方便,如果没有它,就必须计算所需要范围的毫秒数。

在上面的代码中,另一个重要内容是MCI_NOTIFY标记。我们将在"同步命令执行和异步命令执行"一节中学习它。

9.1.4 停止播放

为了停止播放CD,需要发送MCI_STOP消息:

```
void CCDPlayer::Stop()
{
    m_Paused = false;
    if (!m_InitGood) return;
    MCI_GENERIC_PARMS genericparms;
    ZeroMemory(&genericparms, sizeof(MCI_GENERIC_PARMS));
    mciSendCommand(m_DeviceID, MCI_STOP, MCI_WAIT, (DWORD)(LPVOID)
    &genericparms);
}
```

这个带有通用参数的消息是最简单的。这里,代码发送了一个MCI_STOP命令,并提供了一个通用的参数结构。

注意,这里通过设置MCI_WAIT标记来保证在Stop函数返回时,CD播放器已经停止工作。

9.1.5 暂停播放和恢复播放

Play函数启动后所做的第一件事是设置m_Pause标记为假值。由于MCI不支持CD播放器的暂停和恢复操作。因此,为了实现完整功能的CD播放器,不得不自己设计实现暂停和恢复操作的功能。

好在实现并不困难,暂停操纵所要做的操作是查询当前播放位置并记住它,然后发送停止命令。需要添加一个标记来反映CD播放器是否被暂停。而恢复操纵则是从暂停的地方开始

播放,直到音轨结束。

实现这种"伪暂停"方式还需要两个额外的变量。一个是 m_Paused,它是布尔型标记;另一个是 m_PausedPos,它是 CTrackMinSecFrame 类型的,存储 CD 被暂停的时刻。

下面是它的代码:

```
void CCDPlayer::Pause()
{
  if (!m_InitGood) return;
  m_PausedPos = GetCurrentPos();
  Stop();
  m_Paused = true;
}
void CCDPlayer::Resume()
{
  if (!m_InitGood) return;
  if (!m_Paused) return;
  // 从暂停处开始播放,直到音轨结束
  MCIERROR e;
  MCI_PLAY_PARMS mciPlayParms;

  mciPlayParms.dwFrom = 0L;
  mciPlayParms.dwTo = 0L;
  mciPlayParms.dwFrom = m_PausedPos;
  mciPlayParms.dwTo = MCI_MAKE_TMSF(m_PausedPos.m_Track + 1, 0, 0, 0);
  mciPlayParms.dwCallback = (DWORD)m_hWnd;
  e = mciSendCommand(m_DeviceID, MCI_PLAY,
    MCI_FROM | MCI_TO | MCI_NOTIFY,
    (DWORD)(LPVOID) &mciPlayParms);
  if (e) HandleError(e);
  m_Paused = false;
}
```

Pause 函数操作很简单(在下一节学习 GetCurrentPos),而惟一一个关于 Resume 函数的技巧是播放范围。该代码设置了一个 MCI_PLAY_PARMS 结构,以便让 CD 播放器能够从它停止的地方(m_PausedPos)重新开始播放,一直播放到下一个音轨开始处(m_PausedPos.m_track +

1,0,0,0)。最后,它设置 m_Paused 标记为假值,结束暂停操作。

> **提示**
>
> CD 播放器的 Play 方法没有智能。Play 不会自动检测 m_Paused 标记;它无条件地从音轨开始处开始播放。CD 的 Player 类依靠其他的代码来保存暂停状态,并适时地调用 Play 方法或者 Resume 方法。

你会看到另外两个设置暂停标记为假值的方法。比如,弹出 CD 的函数也会设置 m_Paused 为假值。

9.1.6 确定播放位置

为了查询 CD 播放器当前的播放位置,可以发送 MCI_STATUS 消息,MCI_STATUS 可以查询所有的状态。请阅读 MDSN 中这些标记,要查询当前的播放位置,需要传递 MCI_STATUS_POSITION 作为状态参数结构 MCI_STATUS_PARMS 中的 dwItem。

下面是程序代码如下:

```
CTrackMinSecFrame CCDPlayer::GetCurrentPos()
{
  CTrackMinSecFrame pos;
  if (!m_InitGood) return(pos);
  if (m_Paused) return(m_PausedPos);
  if (!IsPlaying()) return(pos);
  MCI_STATUS_PARMS statusparms;
  ZeroMemory(&statusparms, sizeof(MCI_STATUS_PARMS));
  statusparms.dwItem = MCI_STATUS_POSITION;
  MCIERROR e = mciSendCommand(m_DeviceID,MCI_STATUS,MCI_STATUS_ITEM
    | MCI_WAIT,
    (DWORD)(LPVOID)&statusparms);
  if (e) HandleError(e);
  pos = statusparms.dwReturn;
  return(pos);
}
```

要正确确定把一个 MCI 标记放在哪儿有时是比较困难的。在状态消息这种情形下,我发现最直接的方法是指定 MCI_STATUS_ITEM 标记,它告诉 MCI 查看 MCI_STATUS_PARMS 结

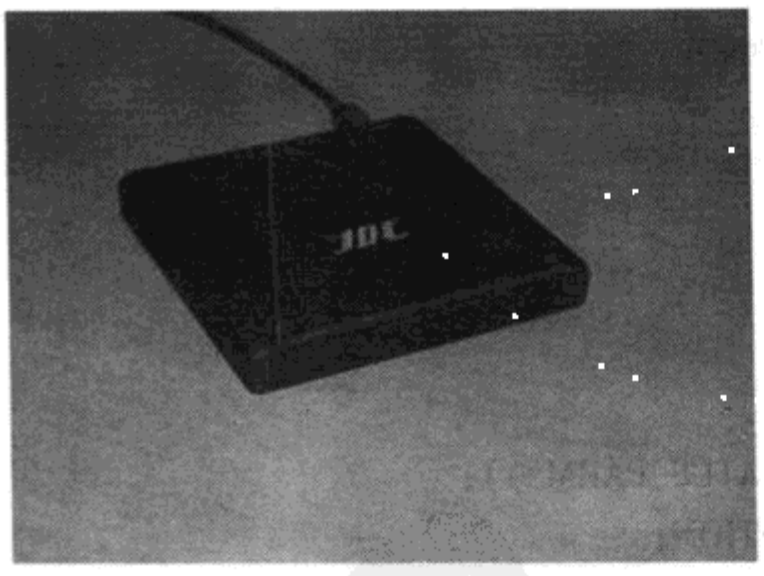
构中的 dwItem 变量,以获得想要的状态项。然后 MCI 把查询的结果放到 dwReturn 变量中,而上面的代码会把结果转换成 CTrackMinSecFrame,并返回。

9.1.7　装卸 CD 托盘

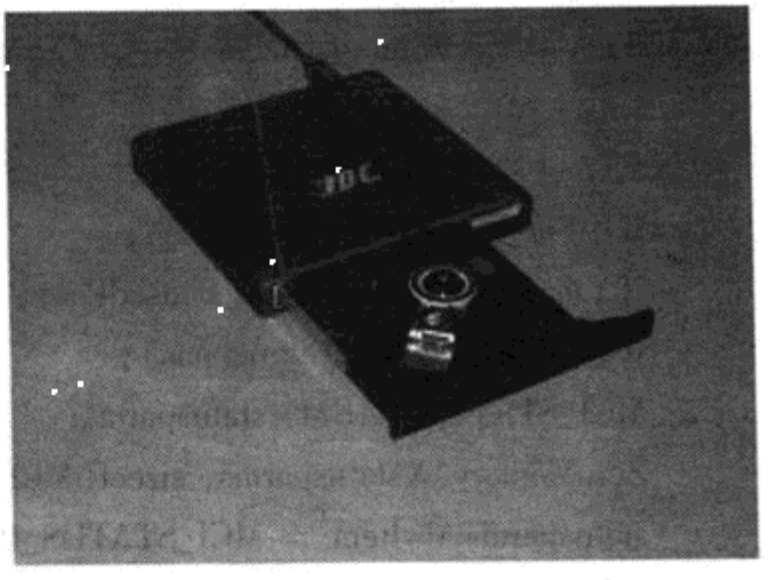
如图 9.2 所示,编写让物体运动的代码是令人兴奋的,这里给出我设计的一段能够让 CD 播放器弹出它的托盘的实用代码。下面是它的具体描述:

```
void CCDPlayer::OpenCDDoor()
{
  if (!m_InitGood) return;
  m_Paused = false;
  mciSendCommand(m_DeviceID,MCI_SET,MCI_SET_DOOR_OPEN,0);
}
```

这段代码再简单不过了。它只是发送 MCI_SET 消息,指定 MCI_SET_DOOR_OPEN 标记就可以了。



在收到MCI_SET_DOOR_OPEN命令以前



在收到MCI_SET_DOOR_OPEN命令以后

图 9.2　发送 MCI_SET_DOOR_OPEN 命令的效果

关闭托盘的仓门也同样简单,其代码如下:

```
void CCDPlayer::CloseCDDoor()
{
  if (!m_InitGood) return;
  mciSendCommand(m_DeviceID,MCI_SET,MCI_SET_DOOR_CLOSED,0);
```

```
}
```

在这儿,MCI_SET_DOOR_CLOSED 标记让 CD 音频驱动器收回它的托盘。如果托盘并没有伸出,进行收回操作也没有关系。

9.1.8 检测 CD 状态

检测 CD 驱动器的状态包括两个内容。第一个内容需要查询设备状态,检测 CD 驱动器中是否有光盘。下面是它的代码:

```
bool CCDPlayer::IsCDInDrive()
{
  if (!m_InitGood) return(false);
  MCI_STATUS_PARMS statusparms;
  ZeroMemory(&statusparms, sizeof(MCI_STATUS_PARMS));

  statusparms.dwItem = MCI_STATUS_MEDIA_PRESENT;
  MCIERROR e = mciSendCommand(m_DeviceID,MCI_STATUS,MCI_STATUS_ITEM |
    MCI_WAIT,
    (DWORD)(LPVOID)&statusparms);
  if (e) HandleError(e);
  return(statusparms.dwReturn == TRUE);
}
```

这段代码发送 MCI_STATUS 消息,查询 MCI_STATUS_MEDIA_PRESENT,如果有 CD 在驱动器中,则返回真值。

第二个内容有点技巧,目的是获得 CD 发生改变的时刻,这里不需要连续地检测 MCI_STATUS_MEDIA_PRESENT。为了正确检测到 CD 所发生的变化,可以监听 Windows 的 WM_DEVICECHANGE 消息。当 CD 盘被取出或者放入的时候,Windows 将发送这条消息到所有程序的顶层 HWND。

> **提示**
>
> WM_DEVICECHANGE 消息在技术上表示硬件配置改变了,因此当连接或者断开 USB 和热插播设备时,你也会得到这条消息。

为了正确解释 WM_DEVICECHANGE 消息,需要查看它的 wParam。如果 CD 盘被取出,wParam 将包含表示“移出设备完毕”的 DBT_DEVICEREMOVECOMPLETE 标记。当一张新的

CD 盘被放入时,将得到 DBT_DEVICEARRIVAL 消息。要了解 WM_DEVICECHANGE 的更多信息,请查询 MSDN 帮助文档。

9.1.9 确定 CD 内容

现在几乎完全可以控制 CD 了,余下的事情是如何确定 CD 盘中的内容? 读者已知道如何得到一张 CD 的产品代码和所生成的 ID,现在需要来关心 CD 盘上有多少个音轨等信息。

这里,将使用 MCI_STATUS 消息,下面是它的代码:

```
int CCDPlayer::GetNumTracks()
{
  if (!m_InitGood) return(false);
  MCI_STATUS_PARMS statusparms;
  ZeroMemory(&statusparms, sizeof(MCI_STATUS_PARMS));

  statusparms.dwItem = MCI_STATUS_NUMBER_OF_TRACKS;
  MCIERROR e = mciSendCommand(m_DeviceID,MCI_STATUS,MCI_STATUS_ITEM |
    MCI_WAIT,
    (DWORD)(LPVOID)&statusparms);
 if (e) HandleError(e);
 return(statusparms.dwReturn);
}
```

> **提示**
>
> 除了我所提到的状态命令以外,还有很多其他的。如果想学习如何查询诸如每个音轨的长度或者音轨的类型(数据或者音频)等信息,请查看 MSDN 文档。

MCI 专门提供了一个状态查询函数 MCI_STATUS_NUMBER_OF_TRACKS,它可以很好地完成查询音轨的任务,并把音轨数量保存到你所给定的 MCI_STATUS_PARMS 结构的 dwReturn 变量中。

9.1.10 同步和异步命令执行

前面,在描述 Play 函数的时候,提到了一个重要标记 MCI_NOTIFY。可以在任何一个 MCI 消息中使用这个标记,因为它告诉 MCI,你想在所请求的事情完成后得到通知。如果把这个标记放到 MCI_PLAY 消息中,当 CD 播放到指定的结束点时,MCI 就会通知播放完成。

MCI 通过把一个 MM_MCINOTIFY 消息传递到主窗口来进行通知。你可以查看 MM_MCINOTIFY 消息的 wParam,确认发生了什么事情,wParam 可能是 MCI_NOTIFY_ABORTED、MCI_NOTIFY_FAILURE、MCI_NOTIFY_SUCCESSFUL 或者 MCI_NOTIFY_SUPERSEDED。lParam 保存了发送消息的设备 ID。

使用 MM_MCINOTIFY,可以循环播放、随机播放以及做其他事情。为了做到这一点,只需要在收到 MM_MCINOTIFY 消息、从 lParam 中得到 CD 播放器设备 ID、或者从 wParam 中得到 MCI_NOTIFY_SUCCESSFUL 时,进行适当的操作。

如果想同步执行一条命令,即想让 mciSendCommand 等到命令完成以后才返回的话,也可以指定 MCI_WAIT 标记。MCI_WAIT 标记使你可以安全地进行操作,并确保一个消息已经处理完成,这在一次发送多条消息时很有用。

同步命令执行过程和异步命令执行过程的比较和区别见图 9.3 所示。更多的信息可查看 MSDN 中的 MM_MCINOTIFY。

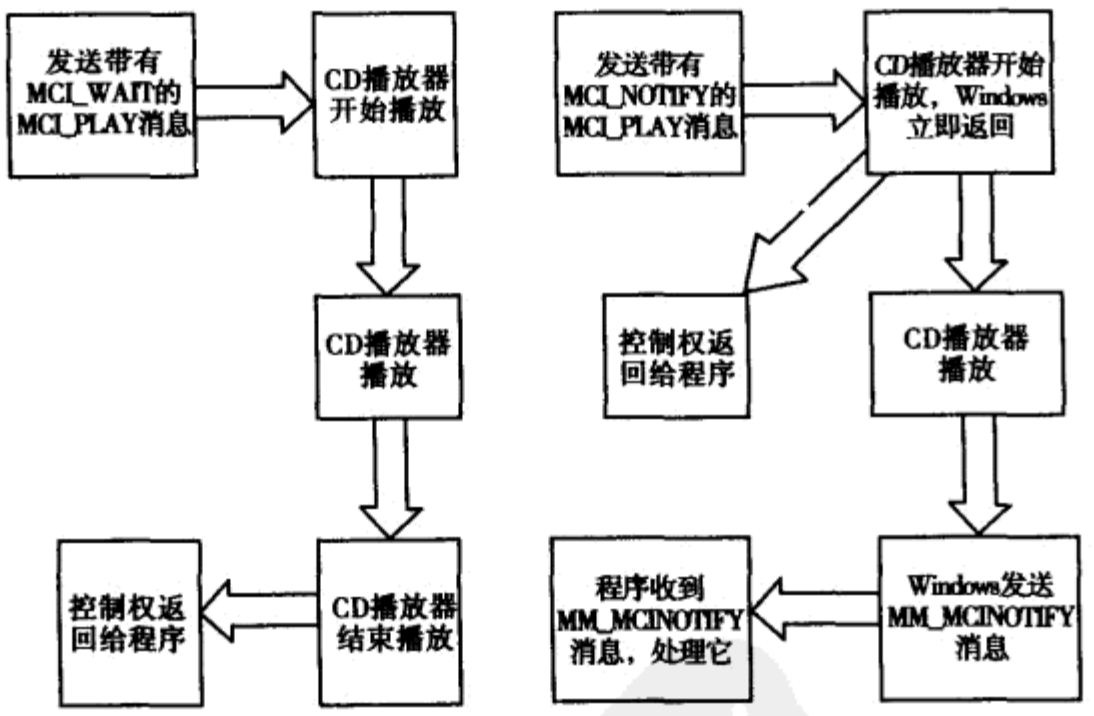


图 9.3 同步命令执行和异步命令执行

9.1.11 CD 音量控制

在第 5.1 节已经讲过,可以使用混音器 API 来控制 CD 播放器的音量,方法是提供正确的行设置,可以使用 CMixer 中预定义的 CDAudio 常量,设置所需要的音量。

9.2　CD 播放器示例程序

本章的示例程序是一个 CD 播放器。这个 Win32 GUI 应用程序使用 CCDPlayer 类,可以播放(暂停等)CD 音乐,它还可以切换不同的音轨,并选择从不同的 CD 设备上播放。图 9.4 是程序的屏幕截图。



图 9.4　本章示例程序运行时截图

Ch9p1_CDPlayer. cpp 中大部分代码是 Win 32 GUI 代码,这是一个 Win32 API 的框架,它没有使用任何基类库。下面将解释一下其中比较有用的地方。

CD 播放器本身是一个全局变量 g_CDPlayer。当主对话框被初始化以后,它先调用 PutAvilableDevicesInComboBox 函数;该函数又调用静态 CCDPlayer::GetAvailableDevices 函数,这个函数遍历整个驱动器字母表,并返回驱动器信息数组;而这个数组则被程序组装到组合框中。

CD 播放器被初始化以响应组合框选择变化消息 CBN_SELCHANGE,将用户选择的驱动器盘符传递给 OnComboBoxChange 函数,重新初始化 CD 播放器。组合框的每一项有 32 位存储空间,示例程序使用其中的 8 位来保存驱动器盘符对应的字符值,因此所作的操作是从当前选中的组合框某项中取出这 8 位数据,然后将驱动器盘符传给 CD 播放器的 Init 函数。

有一个每秒钟执行一次的定时器,叫做 RefreshGUI。这个函数所做的事情包括:①根据选择的驱动器中是否有光盘来激活或者屏蔽控制按钮。②查询 CD 播放器,得到音轨的位置,然后将该信息显示在 IDC_LCD 标签中。此外,它还适时地将播放按钮上的显示文本设置为“Play”、“Pause”或“Resume”。

作者设计了 WM_DEVICECHANGED 和 MM_MCINOTIFY 消息的处理器。设备变化消息处理器 WM_DEVICECHANGED 需调用 RefreshGUI,以便在放入 CD 时激活按钮,而在取出 CD 时屏蔽按钮。当移动到 CD 上的下一个音轨时将产生 MM_MCINOTIFY 消息,可以随意地添加一个循环播放或者随机播放的代码到这个处理器中。

注意,整个对话框过程被放在一个 try 块中。如果 CD 播放器收到任何一个 MCI 错误,就调用受保护的 HandleError 函数,抛出一个字符串。对话框过程捕获这个字符串,并以消息框的形式显示给用户。在实际游戏中,有人可能还希望做其他操作,但对于简单的 CD 播放器,这样做就足够了。

总而言之,如果以前没有做过框架性的 Win32 API 编程,那么学习这个程序会觉得涉及的知识很多。建议读者看一本关于 Windows API 的书,比如《Windows Programming for Dummies》。

9.3 本章小结

不需要深入理解 MCI 是如何工作的，不用考虑 CD 文件系统（该系统相当复杂），也不用考虑流和错误检查，这里所需要做的只是发送 MCI_PLAY 消息，点击虚拟的播放按钮。

当然，读者也可以学得更深入一点，可以学习免费的、开放的程序代码，特别是在 http://www.sourceforge.net 上有一些有用的开放源代码，也可以连接到基于 Internet 的 CDDB 服务（可获取 CD 盘的专辑名称、作者名和音轨名），获得调用低层 ASPI 提取数字音频的开放源代码。本书附带的光盘上包含了这两个链接。

CD 播放器应该可以为你的游戏做任何事情。大部分的游戏只是希望能在某个特定的时间循环播放一段特定的音轨，而这个基于媒体控制接口的类可以完成这些任务。

第 10 章

播放轨迹音乐

在MP3、Ogg Vorbis 和 Windows Media Audio 出现以前，就有了轨迹音乐(tracked music)，见图 10.1 和 10.2 所示。在 20 世纪 90 年代初期，轨迹音乐就被用在了游戏中，并成为最流行的实现游戏音乐的方式，特别是对于需要在线发布的游戏。这是因为它很好地平衡了音质与容量。一个典型的 MP3 文件将占用 3 ~ 4 MB 空间，而轨迹音乐则通常只需要 300 ~ 400 kB，而且音质依然不错。

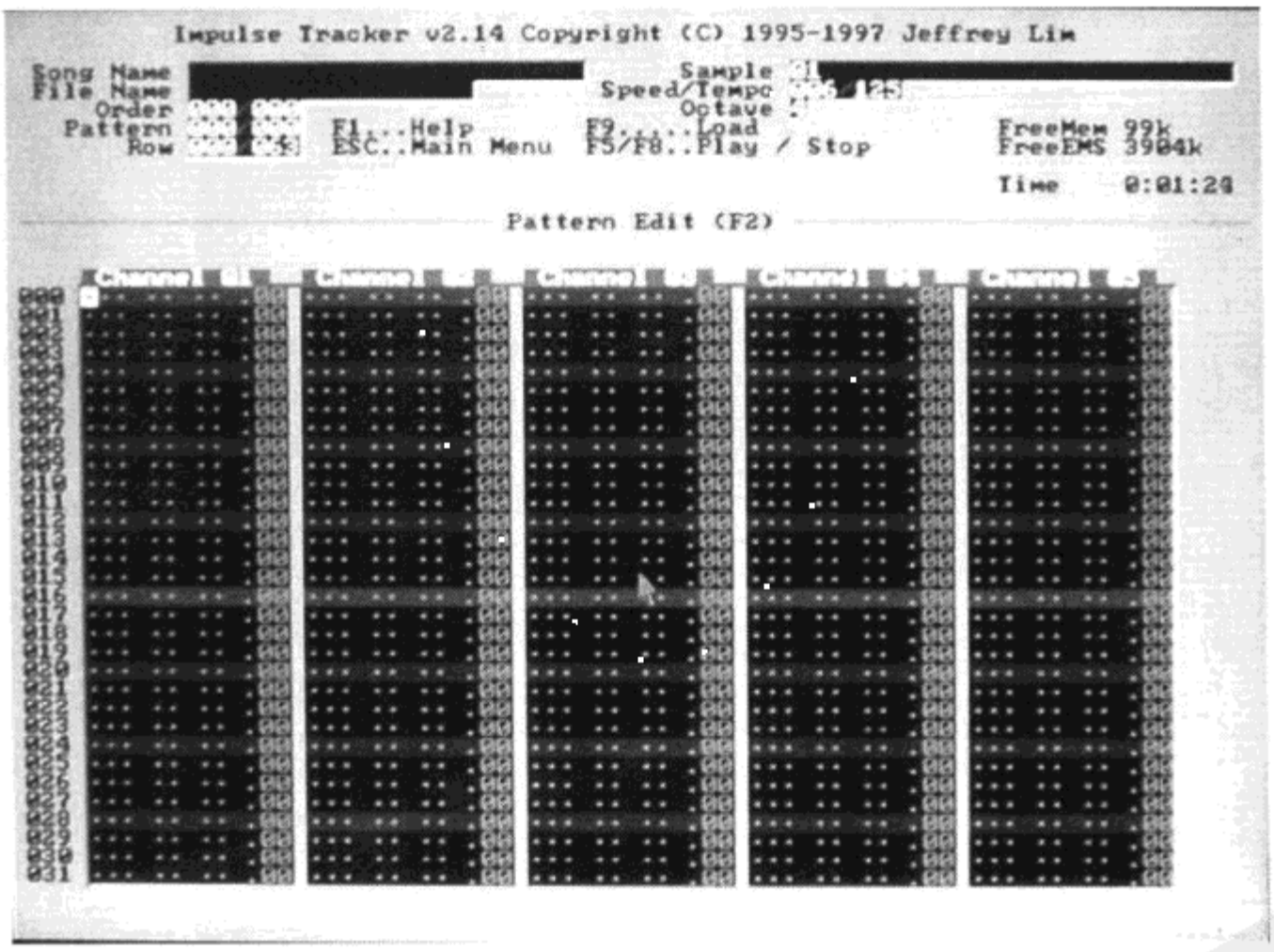


图 10.1 Impulse Tracker，一个常用的轨迹音乐编辑器

(由 Jeffrey Lim 编写)

本章将学习轨迹音乐是如何工作的，以及如何使用 MikMod API 将它集成到音频引擎中，而 MikMod API 是一个 GNU GPL 类库，它可以简化加载和播放轨迹音乐的操作。

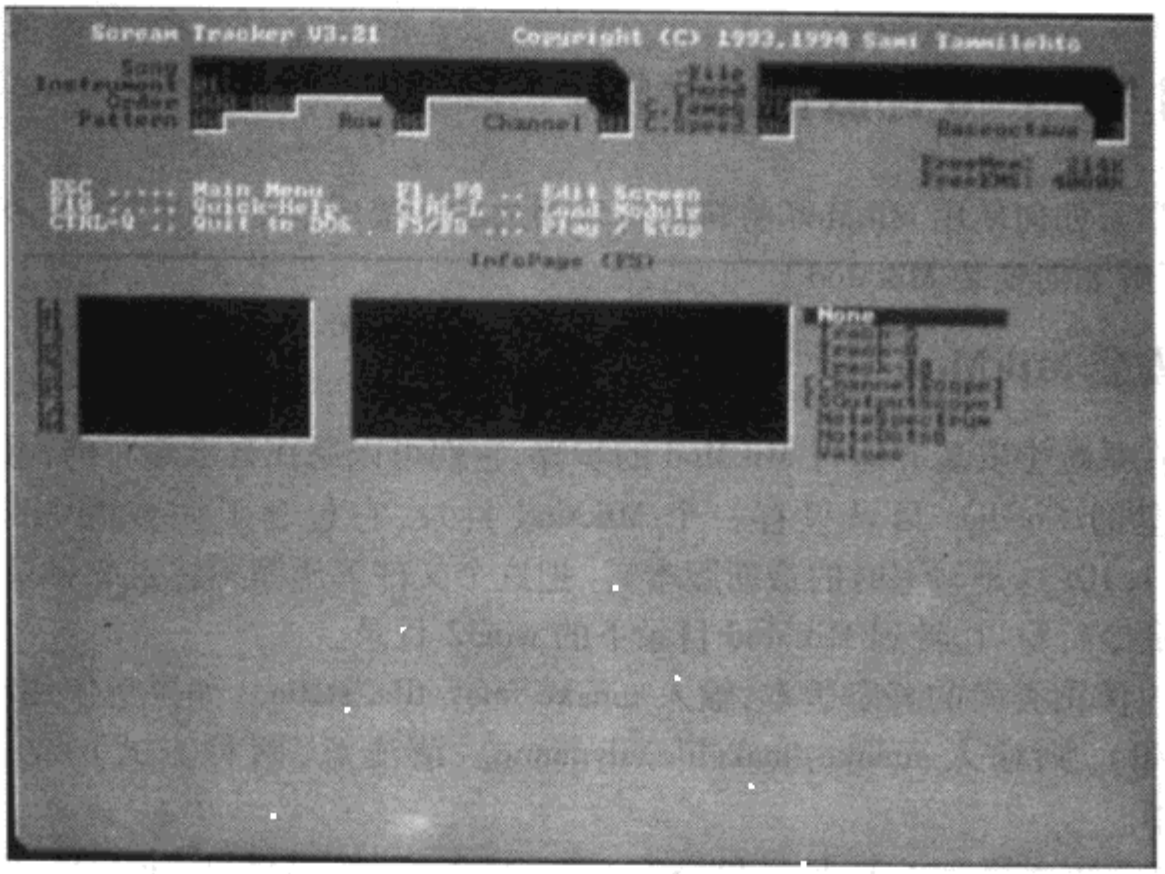


图 10.2　Scream Tracker，另一个常用的轨迹音乐编辑器的截图（由 Sami Tammilehto 编写）

10.1　轨迹音乐基础

以前，很多视频游戏的音乐都使用 FM 合成。通过使用数学公式来调用 FM 合成器生成音阶——例如利用一个正弦波，并加上其他波形以及一些数学效果或者一些噪音来增强效果。用数学方法来表现乐器声音，可以生成范围广泛的不同类型的声音，不幸的是，所有这些声音听起来都留有合成的痕迹。对于某些乐器（像长笛）声音效果比较逼真，但对于大部分乐器而言结果并不理想，对于少数重要乐器（特别是铜管乐器和鼓）声音效果更加糟糕。

轨迹音乐本质上是 FM 合成，但它不使用数学公式来表现乐器的声音，而是将乐器本身的声音数字化后存储在歌曲中，然后 CPU 使用算法对适当的音符升高或者降低样本的音调，从而使得乐器声听起来更接近它们真实的发音。每种乐器的音质虽然并不完美，但当把这些乐器的声音组合在一起时，就可以合成一首很动听的歌。

轨迹音乐在基础层面上做了与 DirectMusic 相同的事情。DirectMusic 合成器允许对乐器使用自定义的样本集（被称为乐队），并提供更多的选项来使用样本。比如，允许一个乐器有多个样本，每个样本所表现的音符有不同的音调和强度。

10.2 用 MikMod 播放轨迹音乐

本节将学习如何使用 MikMod 将轨迹音乐播放功能集成到音频引擎中。在开始学习代码前,首先了解如何安装 MikMod。

10.2.1 安装 MikMod

下面是在系统中安装并运行 MikMod 的步骤,它们可以用在自己编写的代码中。

①在光盘的 Goodies 目录里有一个 MikMod 目录,它包含了完整的 MikMod 软件包 libmikmod-3.1.10(这是写书时的最新版本)。把这个文件解压缩到选定的目录中。

②打开命令行窗口,转到 MikMod 目录下的 win32 目录。

③如果想使用类库的静态版本,输入 nmake-fmakefile. static。如果想使用动态版本(将用 mikmod. dll),则输入 nmake-fmakefile. dynamic。请注意,两种方式的结果都输出到 mikmod. lib 中。

④如果 MikMod 的 include 目录里没有 mikmod. h 文件,请拷贝 win32\mikmod_build. h 到 include\mikmod. h。创建文件本来应该自动做这些操作,但有的系统并没有这样做。

为了在应用程序里使用 MikMod 类库,需要按下面的步骤操作:

①在 DevStudio 中,从 Project 下拉菜单中选择 Settings。

②将 mikmod include 目录添加到项目的 include path 中。实现操作是:转到 C/C ++ 选项,选择 Preprocessor,添加 MikMod include 目录的相对路径。对于本章的示例程序而言,就是. . \. . \libmikmod-3.1.10\include。

③将 mikmod library 目录(实际上就是 win32 目录)添加到项目的 library path 中。实现的步骤是,转到 Link 选项卡,选择 Input as,在 Additional library path 编辑框中添加相对路径。对于本章的示例程序而言,输入. . \. . \libmikmod-3.1.10\win32。

④添加 MikMod 类库到自己的链接类库,即把 mikmod. lib 添加到 object/library modules 编辑框中。

⑤请不要忘记在使用 MikMod 的源程序中包含 mikmod. h。

10.2.2 获得 MikMod 帮助信息

MikMod API 参考包含在 docs 目录里的 HTML 文档中。打开 mikmod. html 文件,它包含了所有帮助文档。

10.2.3 将 MikMod 引入音频引擎

为了将 MikMod 集成到音频引擎中,需要创建一个新类,且需要对 CAudioManager 类作

一些修改。下面分别讨论其细节。

(1) CTrackedMusic

新类 CTrackedMusic 跟其他音乐格式或者引擎中其他音频类一样,是从 CSound 继承而来的,它表示一个轨迹音乐对象。下面是这个类的声明:

```
class CTrackedMusic : public CSound
{
 public:
 friend class CAudioManager;
 CTrackedMusic(CAudioManager *mgr);
 virtual ~CTrackedMusic();
 virtual bool Play();
 virtual bool Stop();
 virtual bool IsPlaying();
 std::string GetFileName() { return(m_FileName); }
 static void FreeAllModules();
 protected:
 static std::vector<CTrackedMusic *> m_AllTrackedMusicInstances;
 std::string m_FileName;
 MODULE *m_Module;
};
```

这个类的核心是指向 MODULE:m_Module 的指针。MODULE 是 MikMod 使用的数据类型,它本质上是一个轨迹音乐歌曲的句柄。通过调用 Player_Load 得到 MODULE 地址。

在 CTrackedMusic 中有另外两个变量。一个是 CTrackedMusic 指针数组,用来保存所有被实例化的轨迹音乐类的指针。在 CTrackedMusic 的构造函数中,代码将它的 this 指针放入数组中;而在它的析构函数中,它从数组中找到并删除它的 this 指针。在引擎关闭 MikMod 前,需释放所有 module。这个机制保证在关闭时所有的东西都被清除。

另一个变量是 m_FileName,它保存了创建 CTrackedMusic 类时所用的文件名。虽然这个变量没有特别用途,但它可以提供一个参考信息。

有 3 个必须实现的函数:Play、Stop 和 IsPlaying,由于使用了 MikMod,它们都只有 1 行代码。对于 Play,只需要调用 Player_Start;对于 Stop,调用 Player_Stop;对于 IsPlaying,则调用 Player_Active。

FreeAllModules 方法遍历整个 m_AllTrackedMusicInstances 数组,调用 module 的 Stop 函数,然后调用 Player_Free 释放掉 MODULE 的指针。

(2) CAudioManager

前面已经介绍了关于 CTrackedMusic 类的细节。为了支持 MikMod,需要对 CAudioManager 作一些小的修改。首先,在 Init 函数中加入如下新的代码:

```
MikMod_RegisterAllDrivers( );
MikMod_RegisterAllLoaders( );
MikMod_Init( "" );
```

这 3 个函数是创建 MikMod 类库时所需要的。MikMod_RegisterAllDrivers 会选择一个恰当的输出驱动程序,在 Windows 平台,可能是 Windows 多媒体音频驱动程序或者是 DirectX 5.0 兼容驱动程序。这意味着可以在不支持 DirectX 5.0 的操作系统上使用 MikMod(如 Windows NT)。

MikMod_RegisterAllLoaders 指定类库所加载的文件格式(IT、S3M、MOD 等)。请特别注意,如果忘了在程序的开头调用这个函数,将不能加载任何文件,而总是得到 Player_Load 返回的 NULL 值。

MikMod_Init 使一切准备就绪,并使用 MikMod_RegisterAllDrivers 所挑选的驱动程序进行初始化和启动。可以在字符串中指定想传给驱动程序的选项。

在初始化函数中有了新的代码,通常意味着在反初始化函数中也会增加新的代码。这里也不例外。下面是新增加的代码:

```
CTrackedMusic::FreeAllModules( );
Player_Stop( );
MikMod_Exit( );
```

第一个函数使所有的 CTrackedMusic 实例释放它们的 MODULE 指针。然后,如果仍然有音乐在播放,UnInit 就调用 Player_Stop,随后调用 MikMod_Exit 关闭所有的对象。正如你所看到的, MikMod 处理相当简单。

10.3 本章小结

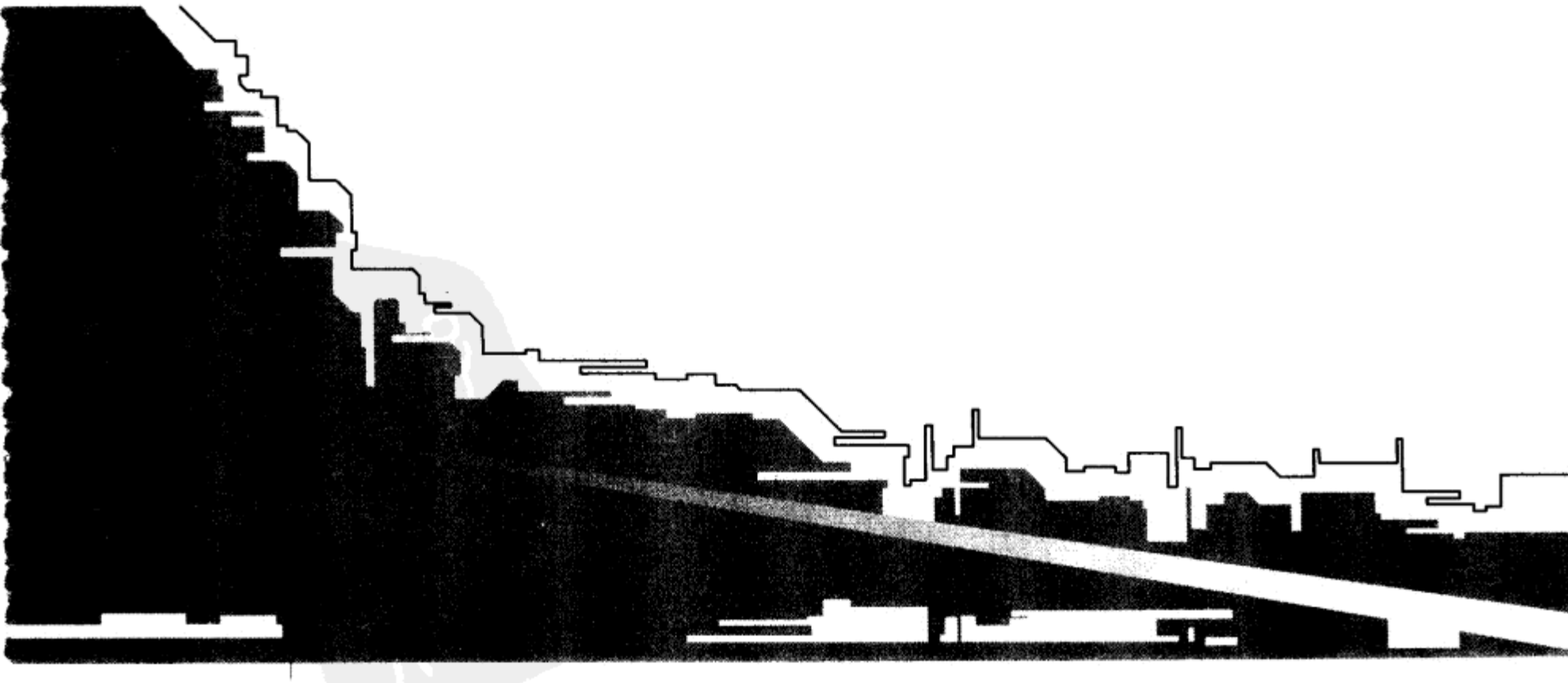
MikMod 使本章操作变得很简单,而本章实际上并没有讲述 MikMod 的全部功能。比如,改变歌曲音乐节拍和音量的 API 命令,请查看 MikMod 帮助文档(docs 文档中的 mikmod. html)获取细节信息。

此外,建议读者阅读 MikMod 的源代码,以便知道在播放轨迹音乐文件时,计算机实际做了哪些事情。

到本章结束,已经学完了音频引擎的所有基本功能。在本书第二部分中,将学习如何结合一些高级技术真正使音频引擎功能强大,比如 3D 声效、音频脚本和效果缓冲区。

第 2 部分

高级音频引擎功能



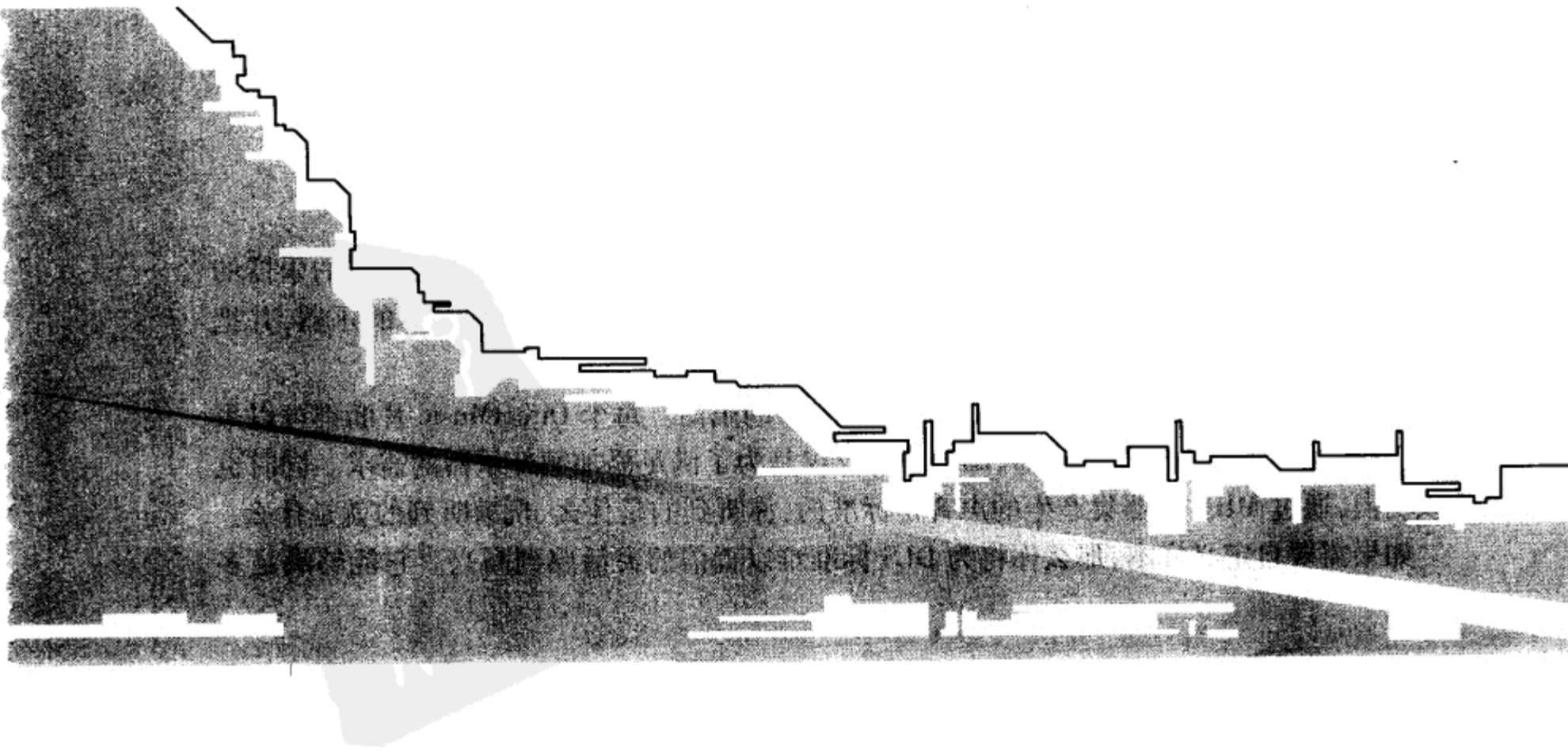
欢迎进入本书高级部分的学习。经过前面章节的学习，读者已经为扩充音频引擎，使其功能不仅仅限于基本的声效和声音播放做好了准备。

在这一部分里，将学习提升游戏音效的几种高级技术并学习如何创建及播放动态音乐（即实时改变的音乐）。动态音乐允许根据游戏事件作出反应，以播放不同的音乐。这一部分还将学习音频脚本，DirectX Audio 的特性允许引擎使用者可以完全控制一个声效或音乐片断的展现方式。有了音频脚本，游戏设计者就可以自己添加、删除或者改变音效，同时不再需要重新编译游戏源代码。

在本部分内容的后半部，将接触到 3D 音效，这种神奇技术将促使听者的大脑去思考声音到底来自于一个 3D 空间中的哪里。使用 3D 音效，当怪物从游戏者身后摸过来时，它的呼吸、脚步声听起来就像真的是从他们后面发出来的。这样，大大增加了游戏场景的真实性和沉醉感。

第 11 章

动态音乐



动态音乐和3D音效是当今视频游戏音效最激动人心的变革之一。数十年来,视频游戏的音乐都是短小音乐片段,不断地重复。我对父母感到歉意,因为在我小学暑假时,他们必须连续数个星期日日夜夜忍受任天堂游戏机一成不变的两分钟循环的8-bit音乐。

今天的计算机已具备根据游戏进程实时合成复杂音乐的能力。因此可以随心所欲地改变音乐或者巧妙地将正在播放的音乐慢慢过渡到其他完全不同的音乐,以增加玩家对游戏的沉醉感。当然,这些都得有一个前提,那就是作曲者和开发者必须紧密协作以创建出让计算机合成动态音乐的材料。

本章和附录A紧密相关。本章中包含了术语,以及关于如何在代码中实现动态音乐的解释。附录A包含了怎样谱曲的例子。为了减少困惑,应该先读本章的第一节(动态音乐基础);然后,像个音乐家似的去读附录A;最后,再以程序员的身份跳回来,学习如何在代码中控制已在附录A中创建的音乐。

11.1 动态音乐基础

在前面几章中,你已经尝到了DirectMusic的甜头,但并没有涉及到深层次的功能。虽然已经知道什么是段,以及如何装载并播放一个段,但并不知道段的里面是什么。

这部分内容术语较多,读者需要做好准备。本节将会涉及到所有需要知道的DirectMusic对象的名字。如果阅读这部分两遍依然弄不懂,请阅读附录A,那里有关于DirectMusic Producer内部所有不同对象的示例。

11.1.1 DLS集合

如果没有乐器,一切都没有意义了。DirectMusic DLS集合是一系列可能用到的乐器的数字样本。当要创建一个新的音乐段时,要做的第一件事就是选出所有需要的乐器,并把它们加入到歌曲的DLS集合中。

必须意识到即使没有DLS集合,也可以创建新的乐谱。每个DirectMusic片断都可以采用MIDI乐器集的所有127个乐器;DLS集合仅仅是为了满足添加额外乐器的需求。同时也应该明白,通过MIDI乐器集产生的声音是标准的,你听到的是什么,玩家听到的就是什么。

如果需要自定义样本,那么你将为DLS标准难以置信的灵活感到高兴。它能够满足多

样化的需求:可以附加多个样本,每个样本用于乐器不同 8 度音阶的发声;而且还可以调整乐器的触发(attack)、衰减(decay)、保持(sustain)、释放(release)(即 ADSR),想要知道更多,请参见附录 A。

11.1.2 音符

当谈到动态音乐时,请注意,输入的音符(note)并非绝对值,这一点很重要。它们与 DirectMusic 播放音乐的当前和弦(chord)和调性(key)相关。这与 MIDI 大不相同,MIDI 中的音符是绝对值。

一个音符包括如下的属性:

- 音调:这是一个显而易见的属性,请记住它是相对的。
- 音量(velocity):它反映了音符的音量,速度的范围从 0(无声)到 127(最大声)。
- 开始时间:音符开始的时间。时间是用小节,拍,1/4 拍(grid),滴答(tick)标定的,方便控制。在 4/4 时间里,每拍被分作四个 1/4 拍。四分音符在指定小节指定节拍的 0 滴答的 0 1/4 拍处开始。八分及十六分音符在不同的 1/4 拍处开始。
- 结束时间:音符结束的时间。同样,这个时间也是用小节,拍,1/4 拍,滴答标定的。
- 持续时间:音符播放的时间长度。持续时间总等于结束时间减去开始时间。

> **提示**
>
> 在 DirectMusic 里,一个音符并不总在准确的时间播放。这听起来骇人听闻,但事实上却是一个重要的特性。DirectMusic 自动地给音符的播放时间制造微小的、随机的延时,以模仿一个人类音乐家。想一想就知道,即使是最好的音乐家也无法把每个音符的时间打准。总会有几个毫秒级的提前或者滞延。DirectMusic 试图通过这样做,以避免音乐过于"完美"和机械。

音符还有一些很少用到的高级属性,比如倒置组覆盖(inversion group overrides)和播放模式。参考 DirectMusic 的帮助文档可获取关于这些属性的更多信息。

11.1.3 轨迹

音符并不是构成轨迹(track)的惟一原料。在一个模式中,还需要有一个音符轨迹(note track)以及一个指示 DirectMusic 播放哪个音符的钢琴键盘。但是不同的轨迹包含不同类型的信息,其作用不亚于音符。比如,风格轨迹(style track)可以告知 DirectMusic 在前 13 小节用一个风格,然后再切换到另一个风格。类似的,一个乐队轨迹(band track)可以告知 DirectMusic 何时改变乐队,而一个和弦轨迹(chord track)可以传达何种和弦适于正在播放的音乐的信息。

11.1.4 乐部

乐部(part)是用同一种乐器演奏的音符集合。如果你谱的是一部交响乐,那就必须为不同乐器(如小提琴、大号、班卓琴等)书写不同乐部,在 DirectMusic 中也一样。

特别地,一个 DirectMusic 乐部包含 3 个不同的轨迹:一个音符轨迹,加上一个变奏切换点(Var swich point)轨迹,以及一个控制曲线(Control Curve)轨迹。变奏切换点告知 DirectMusic 在一个轨迹中何时移向另一个变奏是安全的(从本章后续部分及附录 A 中可以学到更多)。控制曲线轨迹包含了滑音(pitch bends),脚垫移动(foot petal movements),以及当演奏 MIDI 时从键盘获取的调制调整(modulation adjustments)。

11.1.5 演奏通道

可以把一个演奏通道(performance channel)看作是一个乐部和一个音频路径的链接。

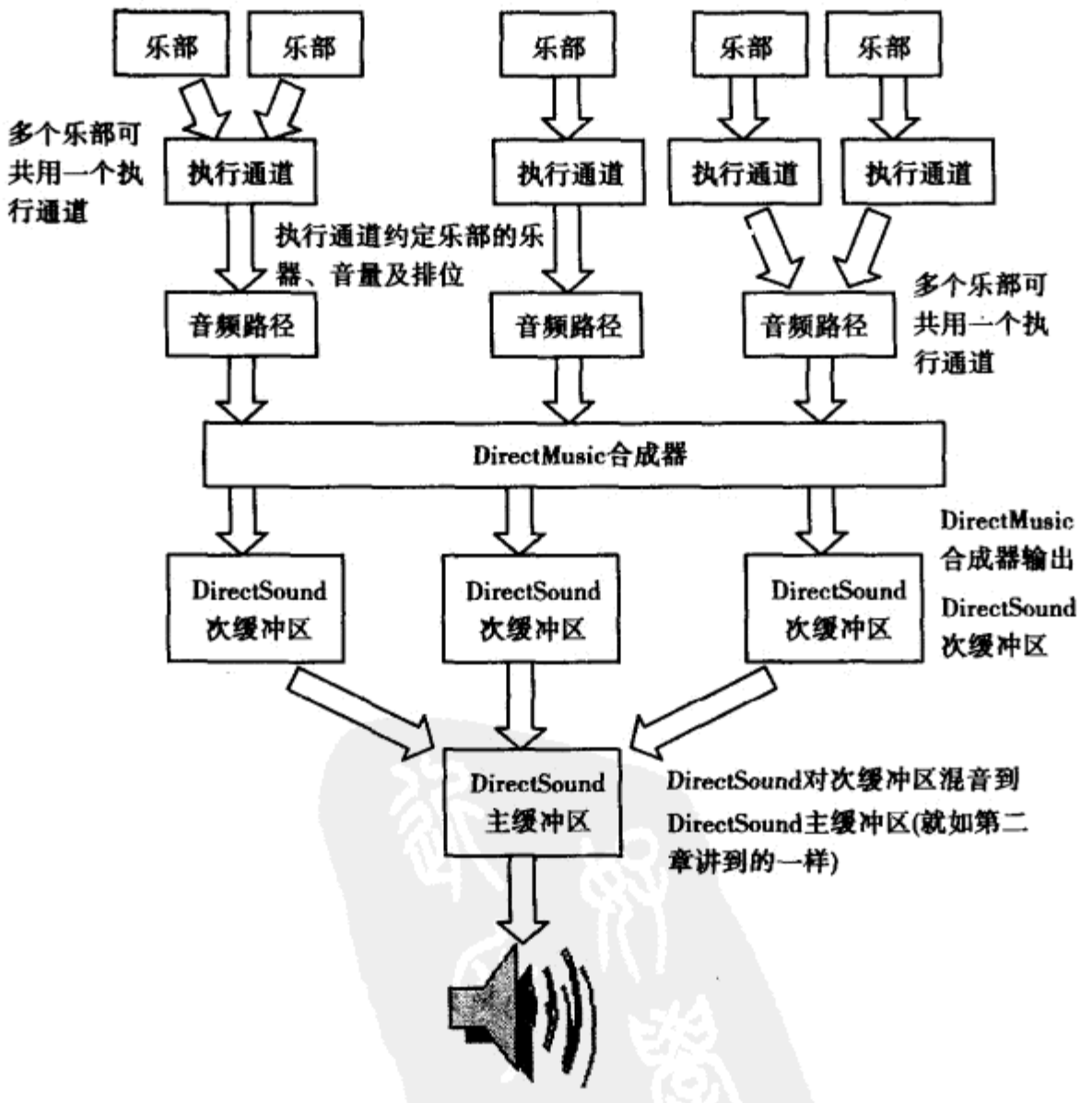


图 11.1 DirectMusic 中乐部、执行通道及音频路径的关系示意图

前面的章节中已学到为每一个想要播放的声音实例创建一个音频路径的方法。音频路径控制声音实例的音量、排位(panning)以及最终声音从哪个输出设备输出。

每一个乐部都有一个输出的演奏通道,见图 11.1。演奏通道指定音量和排位,以及用何种乐队乐器演奏音符。也可以指定在 DirectMusic Producer 中执行的通道用怎样的参数创建相关的音频路径。比如,可以指定一个演奏通道使用混响,使得在此演奏通道上播放的任何乐部都将是混响的。

一个演奏通道可为多个乐部共享。比如,如果有 4 个小提琴乐部并准备以相同的音量和排位用同一个小提琴乐器演奏,则可以让它们共享一个执行通道。演奏通道永远够用,最多可以有 999 个。

11.1.6 模式

模式(pattern)是 DirectMusic 的核心对象之一。模式包含多个乐部,并且可以是所期望的任何长度(通常有几小节,但有时也因音乐结构及个人所期望改变的速度而或长或短)。模式之于 DirectMusic 就相当于“米”之于物理学,“克”之于化学。换句话说,模式是 DirectMusic 工作的习惯单位。如果读者曾经书写过轨迹音乐,那么肯定已经熟悉按模式为单位组织音乐。

> **提示**
>
> 大多数情况下,读者希望自己的模式能够平滑地循环,惟一的例外是修饰模式(embellishment patterns)(后面的小节中会讨论到)。

11.1.7 变奏

每个模式可以有 32 个变奏(variations)。变奏是 DirectMusic 在一个乐部中随机促发的微小差异。这使得你能够保持对音乐的新鲜感。如果把一个模式交给一个音乐家,并告诉他:“请一遍遍重复演奏”,这个音乐家最后会感到厌倦。当他对音乐熟悉后,就会给音乐加入变奏,这里多加一个音符,那里加一个切分音或琶音,从而使其演奏变得更有趣。现在也可以让 DirectMusic 做相同的事,惟一的区别是它不是真正的人类,无法施加个人影响,必须事先提供这些变奏。

注意,变奏不应当是一个模式中完全不同的旋律(虽然可以这样做)。在 DirectMusic 中,可以在很多地方提供众多可选项,并告诉 DirectMusic 随机选择一个。但注意只能在变化较小的可选项处使用变奏;如果有两段完全不同的旋律,应该把它们放在不同的模式甚至不同的风格(style)中,以保持 DirectMusic 对象结构的清晰。

11.1.8 乐旨

可以把乐旨(motif)看作是特殊的模式。乐旨在主旋律发生有趣事件时播放。它们不是由 DirectMusic 随机选择的;相反,其播放时间由你指定。例如,可以在游戏玩家获得能力提升并播放完旋律后,来一小段能力提升小调。

乐旨的正确应用可以使动态音乐听起来很棒。决定在何处播放乐旨的简单办法是看电影。在一部恐怖电影里,当僵尸袭击主角时的音乐增强,或者宇宙飞船逐渐远离发射场时的声音减轻。当然,在电影中这是简单的,因为编导知道有趣的事情何时发生,并把音效部署到合适的时间点。但在视频游戏中,玩家掌控一切,而且不可预测,乐旨必须做好随时播放的准备。

11.1.9 修饰

修饰(embellishment)也被视作一种特殊的模式。当音乐发生显著变化时就会用到修饰,下面所列的是标准的修饰。

- 引子(intro):在动态音乐的开始处播放。
- 间奏(fill):在转接处播放的修饰,如从独唱部过渡到合唱部时。
- 暂停(break):同间奏类似,只是形式不同。一个暂停是指一个或多个乐部暂时停奏以强调其他部分,或者旋律和背景和弦都隐去以突出一段吉他独奏,或者所有的旋律隐声以突出一段鼓曲。
- 尾声(end):动态音乐的快速结尾。

最基本地理解,一个修饰就是一个有特殊标记的模式,以告知 DirectMusic,它是何种类型的修饰。除了上述列出的标准修饰类型外,也可以自定义修饰类型(最多 100 个)。如果有比乐旨更重要的特殊事件需要强调,可以用修饰转移整个模式,以体现更激烈的活动。

在编程者的指示下,DirectMusic 会在合适时机播放修饰,比如在一段音乐的开始、结束或者变调时。

11.1.10 乐队

乐队(band)是指一种风格使用的乐器,它是音乐 DLS 集合的子集,见图 11.2 所示。乐队告知 DirectMusic 使用何种 DLS 乐器,以及乐器的默认排位、默认音量和演奏通道。当你的模式选择“用 3 号乐器发 D sharp”时,DirectMusic 会查看当前的乐队以确定 3 号乐器是什么。

初看起来,为乐器建一层多余的引用没有什么用处,但最后会发现乐队具有非常强的特性,它可以很容易地改变乐器集。每个风格可以有超过一个的乐队,可以在代码中轻松

地切换它们。这样,游戏音效就更能配合游戏事件的变化,从而使游戏更加有趣。

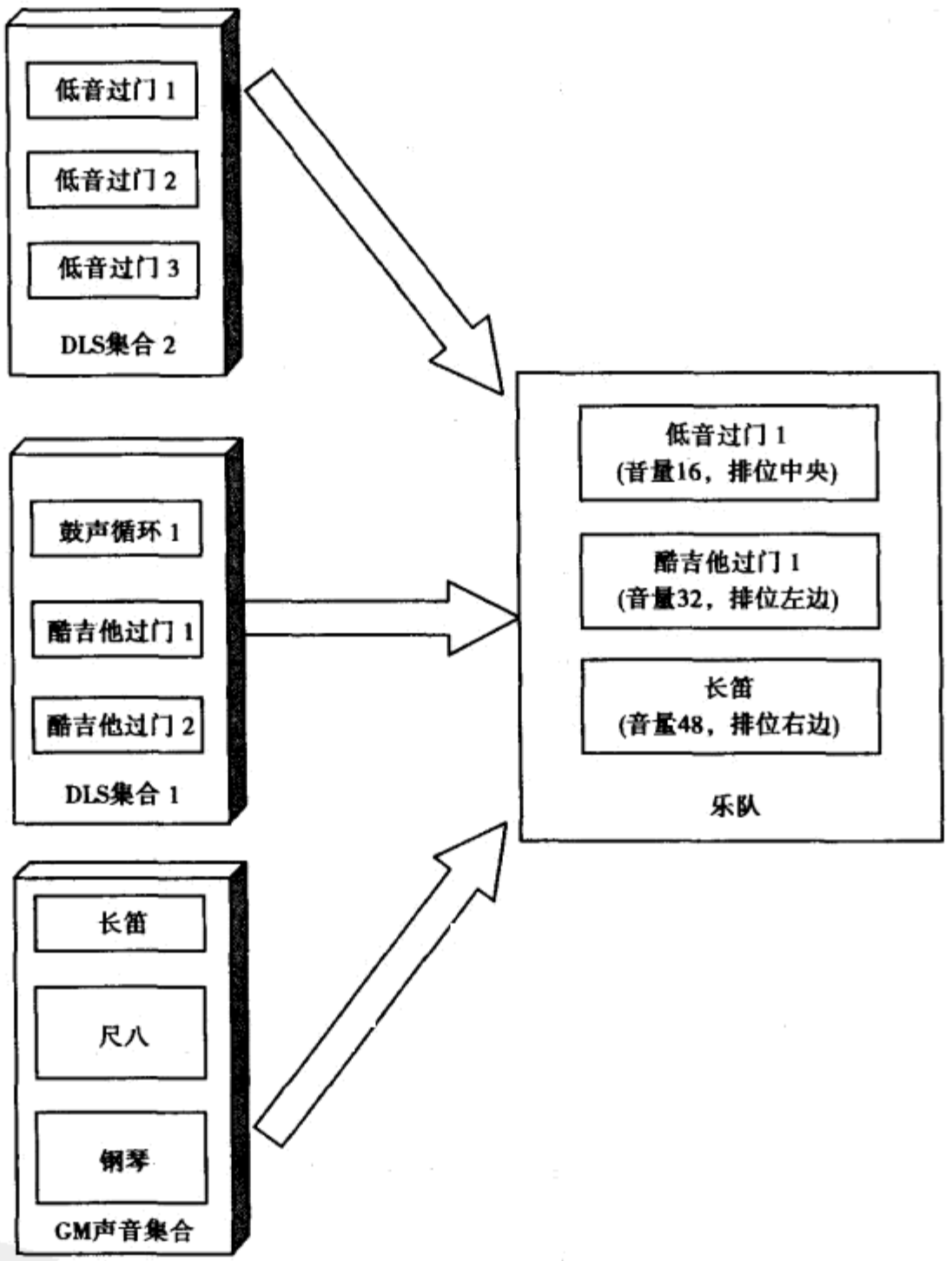


图 11.2 乐队中的乐器来源于 DLS 集合和 GM 声音集合

11.1.11 风格

一个风格(style)是一系列的模式、修饰、乐旨以及乐队构成的。换句话说,一个风格就是一大段音乐。可以把风格当作是胡乱夹满活页乐谱的目录,里面某处可能是一首歌,但是你需要用某种东西(例如段)去组织它。

程序员可以把风格看作是一个程序中的单个 CPP 文件。就如同一个复杂程序的代码

是由多个 CPP 文件组成的一样，一段复杂的音乐也由多个风格组成。

11.1.12 音效段

音效段(segment)是风格的一个集合。段对象中包含的轨迹告知 DirectMusic 何时播放何种风格，以及播放风格中的哪个模式，见图 11.3。

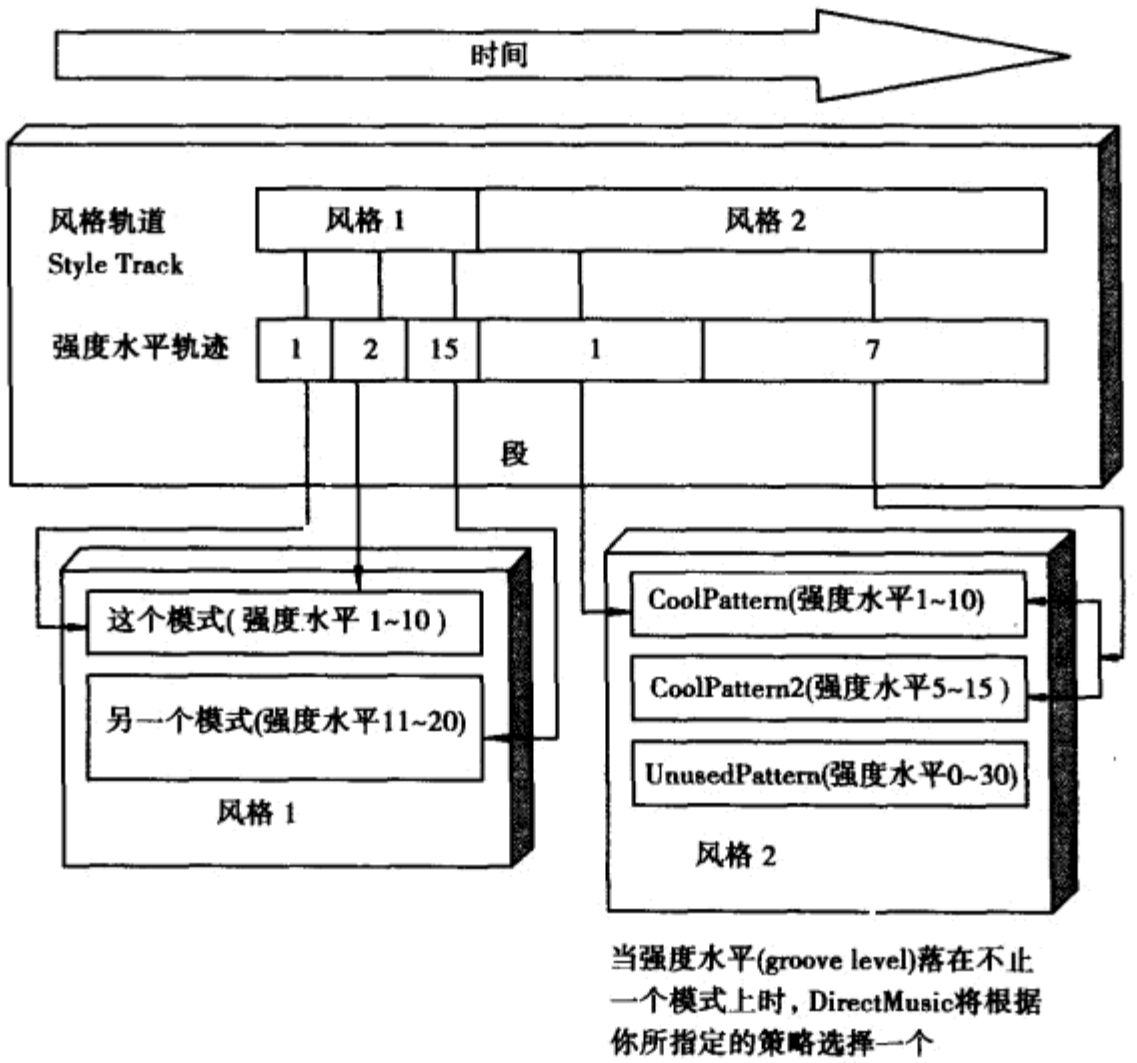


图 11.3 风格和强度级别轨迹告诉 DirectMusic 接下来播放哪个模式

音效段中最重要的两个轨迹是风格轨迹(指向一个特定的风格)和强度级别(groove level)轨迹(帮助 DirectMusic 决定在这个风格中接下来应该播放哪一个模式)。音效段也可以包含乐队轨迹(这样就可以用不同的乐器播放同一个模式)，速度轨迹和其他几个从不同方面影响模式表现的非常用轨迹。

大多数情况下，游戏的一个级别或某个阶段都只有一个音效段。如果正在制作一个具有不同级别的动作游戏，那么最常见的方法就是给不同级别设定不同的音效段。这些音效段可以共享模式、乐队及其他对象。但当你装载游戏级别 1 时，你的代码会自动调用级别 1 的音效段文件(level1.sgt)。

在这种状况下,通常不必担心如何在音效段间转换。可以在级别 1 装载时开始播放 level1.sgt,然后在玩家升级时让它平滑地结束(多半是一段尾声修饰)。

如果你的游戏无法清晰的划分阶段,那么就需要在音效段间切换。例如,在一个空地格斗游戏中,你也许会选择给不同的游戏玩家心理状况创建不同的音效段。当玩家开始游戏时,播放"空闲,到处看看"段;当玩家遇到敌人时,也许会跳到格斗段;当老大进来时,也许会从格斗段切换到老大段;然后如果玩家打败了老大,再切换到空闲段。或者,在多玩家的死亡竞赛游戏中,你或许会为玩家的输赢设置不同的音效段。

正如所看到的一样,切分音乐的方法有无限多种。笔者本人喜欢应用这样的原则:一个音效段是一段自包含的音乐,自成一体并能产生变化以使事物更加有趣。当然,把所有的游戏音乐都放在一个音效段里,或者把音乐的每一个小节都切分成一个音效段,都是可能的,但无疑这两种方法都太极端了。

11.1.13 强度及强度级别

每一个模式都会有一个强度(groove)范围,从一个最小值到一个最大值,而两者都将在 1 ~ 100 之间。默认情况下,一个模式的强度范围为 1 ~ 100,但可以另外设定到任意区间,如 10 ~ 20,甚至是一个单一的数字。

DirectMusic 应用强度范围确定播放哪一个模式。每一个音效段都包含一个强度轨迹,以告知 DirectMusic 当前的强度级别是多少,并最终确定哪些模式可以被播放。例如,如果当前的强度级别是 10,而你有两个模式:模式 A 的强度范围为 1 ~ 5,模式 B 的强度范围为 8 ~ 14,DirecMusic 会选择模式 B 播放,因为当前强度级别落在其强度范围之内。

强度轨迹也负责告知 DirectMusic,当有多个模式可供播放时如何处理。比如,假设上例中模式 A 的强度范围为 9 ~ 11,这意味着模式 A 和 B 都可以被播放。强度轨迹通过额外的参数以指示 DirectMusic 在两者之间随机选择(默认行为)或者先播放 A 再播放 B,或者更广泛的其他处理方法。
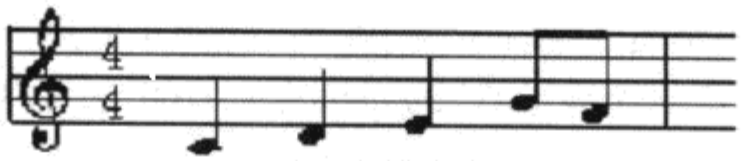
> **提示**
>
> 强度范围并不一定是一个范围。即,你可以指定其最低及最高数字为同一个数,意为只有强度级别刚好等于那个数字的模式才可以被播放。

在代码里,通过使用全局强度级别参数,可以调整当前的强度级别。这个参数与段中的强度级别相加以确定最终的强度级别。例如,如果音效段的强度轨迹设定强度级别为 3,而全局强度级别为 7,则 DirectMusic 会试图播放强度范围包括 10 的那些模式。

11.2 轨迹音乐与 DirectMusic 术语对照

现在,已经描述了如此多的术语,读者也许正在思考如何把它们关连起来。为了解决这个问题,本节将从另一个不同的视角来讲述 DirectMusic:把它同轨迹音乐进行比较,轨迹音乐(IT,S3M 等)的对象层次如下所述:

- 行(rows):指可以置放音符的地点。行在大多数情况下是空的,但有时也包括一些音符、音量、效果或者命令。比如,G01 意为"不要直接播放这个音符,要从最后一个音符慢慢滑向它"。
- 轨迹(tracks):也可称为列或声音,轨迹事实上是谱线空白的数组。一个轨迹通常有 64 个谱线空白,但一些编辑器允许你改变一个模式的轨迹长度,如脉冲编辑器。
- 模式:也称为表格,是音乐最核心的组成部分。一段轨迹音乐至少包含一个模式,通常是多个。一个 64 行的模式通常被分成 4 个小节,每节 16 拍。每隔 4 行放置一个



五线谱中的音符序列

	Channel 01
000	C-5 01 60 .00
001	00
002	00
003	00
004	D-5 01 60 .00
005	00
006	00
007	00
008	E-5 01 60 .00
009	00
010	00
011	00
012	G-5 01 60 .00
013	00
014	F-5 01 60 .00
015	00
016	00

相同的序列在一个典型记录器中的显示

图 11.4 轨迹音乐及五线谱中的音符序列

音符，你便有了一个基本的四分音符，4/4 拍节奏；每隔 2 行放置两个音符便得到一个八分音符，如图 11.4 所示。

- 顺序表(Order List)：也称作模式表或模式序列，它指定模式播放的次序。
- 样本表：这个表保存所有在歌曲中用到的数字样本。在歌曲完成之前，样本表中的每条记录都含有乐器的名称，完成之后，合成器将重写这些名字使它们构成几段文本。
- 样本参数：每个样本的参数(默认音量及排位)。

值得高兴的是，所有这些对象(术语)在 DirectMusic 中都有对应。以下是关于这些核心轨迹音乐对象与 DirectMusic 对象的差异的描述。

11.2.1 行对应于音符

在 DirectMusic 里，并没有“行”的概念。相反，每个音符都附带一个时间戳及一个持续时长参数，以告知 DirectMusic 应在何时播放并持续多长时间。这种时间戳的分辨率是相当精确的，足以支持一些诸如三连音符、奇数时间签名等不可思议的事情。

DirectMusic 把时间分成小节、节拍、1/4 拍(grid)、滴答(tick)等单位，以及第五个可选的 +/- 误差参数，以指定音符可以提前或延后多少滴答被播放。例如，如果我有一个音符，在第一小节，第一拍，第一个 1/4 拍的第 50 滴答开始(奇音符)，且设定 +/- 参数为 10，则 DirectMusic 会在 40 滴答到 60 滴答间的某一时刻开始播放音符。这仅仅是一个例子，在现实生活中，99.9% 的音符都会从 0 滴答开始，因为确实很少需要那么精确的控制。每个音符的持续时长也有一个类似的 +/- 误差参数，以使其长度随机变化。

音符也包含音量，只不过现在它们称为音速(velocity)。同样，有一个类似的 +/- 误差参数用以描述速度范围。

11.2.2 列对应于与乐部

在上一层来看，轨迹音乐的列就相当于 DirectMusic 的乐部。每一种乐器一个乐部，所以一个弦乐四重奏应该有 4 个乐部。在 DirectMusic 里，只能为每个乐部指定一种乐器，不能让一个乐部包含多个乐器的声音。在轨迹音乐中，会把多个乐器叠加在一个轨迹上，因为轨迹是非常稀缺的资源。但对于 DirectMusic 而言，它可以提供成百的乐部，没有必要把多个乐器堆在一起。

11.2.3 模式还是模式

再往外一层，我们看到模式。轨迹音乐的模式同 DirectMusic 的模式是相似的。但又不完全相同。在轨迹音乐里，模式通常很短(最大 4 小节，如果音乐很和缓，速度较慢)，但

DirectMusic 允许模式长达若干小节。对于运行于 8bit 控制台系统的游戏(比如任天堂)来讲,一个好的原则是让模式同背景音乐一样长,换句话说,30 秒 ~ 1 分钟。风格 A 里 4 ~ 8 个小节的内容,接下来是风格 B 中的 4 ~ 8 个小节的内容,最后再循环回风格 A。当然,没有任何条款强制这么做,完全可以一小节一个模式,也可以把整首歌都放进一个模式。

好的动态音乐应该为不同的强度级别(即强烈程度)指定多个不同的模式。

11.2.4 顺序表对应于强度和风格轨迹

现在来看对比差异最大一组对象。在 DirectMusic 里,不存在像顺序表(或模式表)这样的对象。相反,在音效段里有两种不同类型的轨迹。其中,风格轨迹告知 DirectMusic 音效段里有哪些模式可供选择(记住,一种风格是一系列模式的集合),而强度轨迹通过描述强度级别以缩小可选模式范围。

例如,我们假定音效段的风格轨迹是恒定的,且指定 MyCoolStyle. stp 作为整首歌的风格。在 MyCoolStyle. stp 里有两个模式。模式 A 的强度范围为 1 ~ 60,模式 B 为 50 ~ 100。当音效段的强度轨迹指定强度级别为 1 时,仅有模式 A 可以被播放,因为模式 B 的强度级别相对于当前强度级别太高。相反,如果音效段强度级别轨迹指定强度级别为 75,则仅有模式 B 可被播放,有趣的事情发在生强度级别为 55 时,两个段都可以被选择,DirectMusic 此时会随机选择一个播放。

如果你还感到迷惑的话,通过下列方法,可以把 DirectMusic 当作轨迹音乐一样来写:

①为每个模式指定一个惟一的强度级别。即指定一个数字,而不是一个范围。第一个模式的强度级别为 1,第二个模式为 2,以此类推。

②给音效段添加一个风格轨迹,并指定其总是播放同一个风格(当所有的模式都保存好之后)。

③现在,可以把强度级别当作顺序表使用。在强度轨迹中,每指定一个新的强度级别,相应的模式就被播放。这样与轨迹音乐并不是完全不同的,因为强度轨迹可以在任意时间切换模式,而不仅仅是在模式的边界,但这已经很近似了。

④如果想玩点花样,可以给多个不同的模式设定相同的强度范围,DirectMusic 会在它们之间进行随机的选择。

事实上,情况要复杂一些。当改变段的强度轨迹的强度级别时,可以指示 DirectMusic 在有多个模式可供选择时应当如何处理。通过如图 11.5 所示的对话框,可以指示它按如下策略之一处理:

- 随机选择一个模式,播放;再随机选择另一个,播放;直到强度级别被改变。这是默认行为。
- 随机选择一个幸运的模式,然后一直重复播放直到强度级别被改变。这称为

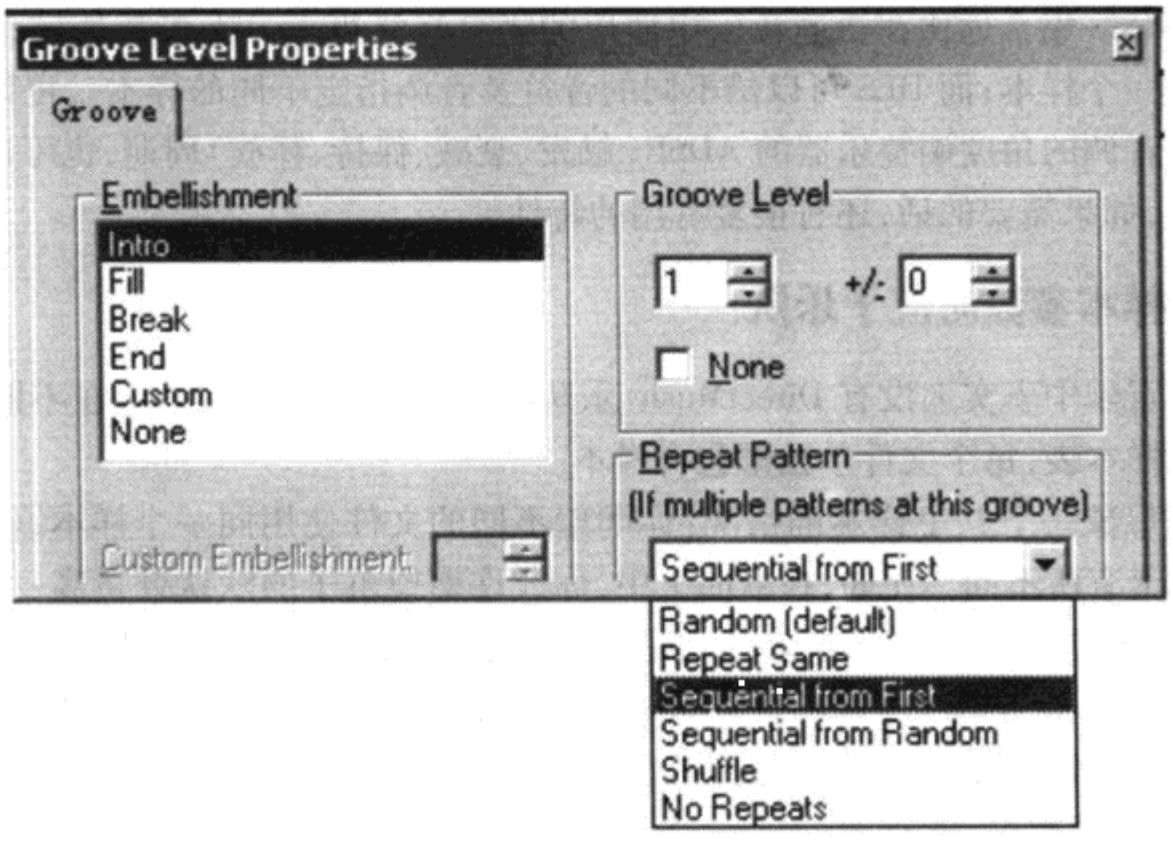


图 11.5 强度级别属性窗口

"Repeat Same"。

- 从第一个可选模式开始,按顺序播放每一个模式。然后循环到第一个模式,直到强度级别改变。这称为"Sequential from First"。
- 随机选择一个模式并播放,接下来顺序播放后续模式,如果必要的话就循环直到强度级别改变。这称为"Sequential form Random"。
- 把所有模式打乱成一个随机顺序,并按照那个顺序播放并重复直到强度级别改变。这称为"Shuffle"或者"No Repeat",这取决于你是否希望在所有模式都被播放完之前重复某个模式。

11.2.5 样本表对应于 DLS 集合

要创建任何轨迹音乐,都必须有乐器样本。但在 DirectMusic 里,不需要提供 WAV 文件。所有的 DirectMusic 谱曲都可以默认采用通用 MIDI 声音集合。通用 MIDI 声音集合包含 127 种不同的乐器,从钢琴、小提琴到 5 孔竹笛(日本的一种乐器)及各种音效。这些高质量的样本内置于 DirectMusic 中,你可以确信它们在所有平台上都会发出相同的声音。

这个声音集使得不使用任何自定义样本的 DirectMusic 乐谱变得平常。然而,这个声音集不支持设计者在乐曲中使用诸如鼓声循环、抒情诗、戏谑吉他过门(wicked guitar licks)等效果。这是 DirectMusic 为什么支持 DLS 集合的原因。

DirectMusic 集合远比在轨迹音乐中使用的集合高级得多。大多数轨迹音乐格式只允许一个乐器一个样本;而 DLS 可以给不同的音符及音调指定不同的样本。可以同时从相对音量和相对音调的角度调整乐器的 ADSR(触发、衰减、保持、释放)周期,也可以创建鼓集合(drum sets),如果需要的话,还有很多有用的特性。

11.2.6 样本参数对应于乐队

在轨迹音乐中事实上没有 DirectMusic 乐队的对应概念。因为无法在不同的轨迹音乐文件间共享样本表,每个文件单独都包含样本。

我们来想象一下共享样本的情况:你期望不同的文件使用同一个样本,仅仅是在音量和排位参数上有所不同。比如,在歌曲 1 中,你或许期望笛子的默认音量高一些,因为它表达旋律;而在歌曲 2 中,由于笛子表达的是背景和弦,你则会期望其默认音量低一些。这就是 DirectMusic 引入乐队的原因。在 DirectMusic 中,不同的歌曲共享相同的样本不仅可能,而且十分普遍,至少,每个人都使用通用 MIDI 声音集。你需要一个地方以表达如下内容,“这首歌用到 5 孔竹笛,音量大一些,并主要从左扬声器中发出”,而另一首歌或许期望它用一半的音量从右扬声器播放,乐队提供了这种调整的灵活性。

11.2.7 结论:DirectMusic 优于轨迹音乐

我们已经看到,DirectMusic 和轨迹音乐有很多概念相似,这一节的目的是为给轨迹音乐专业人员学习奠定基础。不要误认为 DirectMusic 仅仅是另一种类型的轨迹音乐。不论从术语还是过程看,用 DirectMusic 写动态音乐和用轨迹音乐写都有很大区别。花点时间学习 DirectMusic,在为游戏编写动态音乐时,就会得到补偿。

11.3 动态音乐编程

如果是第一次接触动态音乐,建议先停止阅读本章并跳到附录 A,附录 A 会带领你创建一个简单的动态乐谱。当创建好乐谱后,再回到本章学习如何书写配套的代码。

11.3.1 创建 CDynamicMusic 类

同前面的代码一样,本章的代码也从创建一个新类开始。这个新类是 CDynamicMusic,它继承于已介绍的 CdirectMusicSegment 对象类。

受益于良好的 C ++ 设计,本章的代码非常简单。因为由 DirectMusic Producer 生成的乐谱就由音效段构成,CDirectMusicSegment 将会完成其中大部分的工作,你所要做的事就是给 CDirectMusic 类添加几种 DirectMusic 的特殊操作。

11.3.2 装载动态音乐

以下是为初学者准备的,从磁盘、内存或者其他资源装载动态音乐段的代码。(这些代码现在已经显得陈旧过时。)

```
CSoundPtr CAudioManager::LoadDynamicMusic(std::string filename)
{
 CDynamicMusic *music = new CDynamicMusic(this);
 LoadSegmentFromDisk(music, filename);
 music -> LoadMotifs();
 return(CSoundPtr(music));
}

CSoundPtr CAudioManager::LoadDynamicMusic(unsigned char *data, int datalen)
{
 CDynamicMusic *music = new CDynamicMusic(this);
 LoadSegmentFromMemory(music, data, datalen);
 music -> LoadMotifs();
 return(CSoundPtr(music));
}

CSoundPtr CAudioManager::LoadDynamicMusic(HMODULE hmod, char *type, WORD
resID)
{
 CDynamicMusic *music = new CDynamicMusic(this);
 LoadSegmentFromResource(music, hmod, type, resID);
 music -> LoadMotifs();
 return(CSoundPtr(music));
}
```

这段代码与过去的惟一区别是它先创建一个 CDynamicMusic 对象类,然后调用其 LoadMotifs 函数;调用完后返回一个 Csound 智能指针。

LoadMotifs 装载所有音效段里的风格所包含的乐旨。CAudioManager 会在歌曲装载之初调用这个函数。当需要播放乐旨时,代码不需要浪费时间去寻找乐旨所特定的接口 IDirectMusicSegment8。LoadMotifs 会遍历整个段,并为 IDirectMusicSegment8 接口整理出乐

旨名称的映射,编码如下:

```
void CDynamicMusic::LoadMotifs()
{
 IDirectMusicStyle8 * pStyle = NULL;
 DWORD dwStyleIndex = 0;

 while(1)
 {
  HRESULT hr = GetSegment() -> GetParam( GUID_IDirectMusicStyle,
      0xffffffff, dwStyleIndex, 0, NULL, (VOID * )&pStyle);
  if(FAILED(hr)) {
    // 并非一个致命错误……仅表示没有更多的风格
    break;
  }

 DWORD dwIndex = 0;
 while(1)
 {
  WCHAR wstrMotifName[MAX_PATH];
  CHAR  strMotifName[MAX_PATH];
  if(S_FALSE == pStyle -> EnumMotif(dwIndex, wstrMotifName)) {
    // 此风格中没有更多的乐旨
    break;
  }

  wcstombs(strMotifName, wstrMotifName, wcslen(wstrMotifName) + 1);
  CDirectMusicSegment * newseg = new CDirectMusicSegment(m_Manager);
  hr = pStyle -> GetMotif(wstrMotifName, (IDirectMusicSegment
      * * )&newseg -> m_Segment);
  ThrowIfFailed(hr, "CDynamicMusic::LoadMotifs: pStyle -> GetMotif() failed.");
  // 将此段加入到映射中
  m_MotifMap[std::string(strMotifName)] = newseg;
  dwIndex ++;

 }
```

```
        }
        SAFE_RELEASE(pStyle);
        dwStyleIndex ++;
      }
    }
```

这段以 PlayMotif DirectX 示例程序代码为基础的代码，遍历了段中所有的风格；对于每个风格，它又再次遍历风格中所有的乐旨；对于每个乐旨，会获取一个 IDirectMusicSegment8 接口，并赋予此接口一个新的 CDirectMusicSegment 类；然后再把新类放到 m_MotifMap 映射中。

下面给出的代码是 CDynamicMusic 的析构函数，它小心翼翼地删除所有由 LoadMotif 创建的新类 CDirectMusicSegments：

```
CDynamicMusic::~CDynamicMusic()
{
  for (std::map<std::string, CDirectMusicSegment * >::iterator
       i = m_MotifMap.begin(); i != m_MotifMap.end(); ++i) {
    delete (*i).second;
  }
  m_MotifMap.clear();
}
```

这个析构器使用一个迭代器(iterator)遍历映射的所有内容，并同时将遇到的每个 CDirectMusicSegment 删除。

11.3.3 播放动态音乐

为了让动态音乐正常工作，必须改变 CDirectMusicSegment 中的一项重要内容。要知道为什么，必须先理解什么是控制段。

DirectMusic 允许同时播放多个段。比如，可以一边播放动态音乐段，一边播放一个 WAV 段作为声效。

正如你所知道的一样，每个段都包含多个轨迹。如果同时播放两个段，而每个段都包含它们自己的强度轨迹，这没有什么关系，因为大多数的轨迹都只影响它所在的段。但是，段的某些轨迹需要在全局发挥作用，比如，所用的段都必须共享相同的速度和和弦地图(chord map)(因为播放两个不同速度、不同声调的音乐是没有意义的)。所以，如果有一个 2CM 和弦的段和一个 A#m7 和弦的段，并同时播放它们，DirectMusic 如何决定选择哪个段的和弦轨迹呢？这正是控制段需要解决的问题。控制段凌驾于其他所有段之上，

DirectMusic 始终选择应用它的全局轨迹(和弦、速度等)。

通常可以把动态音乐段作为控制段。如果不指定控制段,则应用的所有变奏都不会发生作用,音乐将无法正确播放。

可以方便地告知 DirectMusic 哪个段是控制段:在调用 IDirectMusicPerformance8 接口的 PlaySegmentEx 函数时向其传递 DMUS_SEGF_CONTROL 标志。但由于音频引擎的 C ++ 结构,要让这个选项可用,需要修改几个不同的类。

从最高层开始,需要为 CDirectMusicSegment 的 Play 函数增加一个布尔值参数。这个新加的参数取名为 control,用于表示所播放的段是否作为新的控制段。

这个新参数将被直接传递给 CSoundInstance 的 Init 函数,因为调用 PlaySegmentEx. Init 的 CSoundInstance: Init 也会有一个新参数(仍取名为 control),这个参数将决定是否给 PlaySegmentEx 函数传递 DMUS_SEGF_CONTROL 标志。

现在,重定义 CDynamicMusic 的 Play 函数以应用新标志:

```
bool Play( ) { return( CDirectMusicSegment::Play( NULL, true) ); }
```

这虽简单,但很重要,因为它能保证动态音乐被正确地播放。

11.3.4 改变强度级别

现在读者已经了解如何播放动态音乐,接下来学习如何让动态音乐适应当前环境。我们假设游戏角色跳到一只小恐龙身上,且希望在此时改变音乐。通过设置强度级别则可以完成这件工作。假设创建了两个“没有恐龙”的模式,并设定它们的强度级别为 1 和 2。然后你创建两个“有恐龙”的对应模式,并设强度级别为 11 和 12。而在音效段里,创建一个强度轨迹在强度级别 1 和强度级别 2 间切换,所以两个模式可以轮流切换。(这些内容在附录 A 中有更详细的阐述)。

在游戏中,当角色跳到恐龙背上时,只需把播放强度级别编辑器设为 10,DirectMusic 会把这个 10 同当前强度级别加起来。根据当前所在段中位置的不同,当前强度级别为 1 或者 2,这就意味着,将获得值为 11 或者 12 的强度级别,一个“有恐龙”的模式被播放。

当角色中恐龙消失时,代码将播放强度级别编辑器设置为 0,这将使强度级别变回到 1 或者 2,没有恐龙的模式将会被播放。

刚才所描述的例子仅仅是 DirectMusic 一个简单明了的变化。也可以让 DirectMusic 不要如此剧烈的变化,即让代码随时间慢慢地改变播放强度级别编辑器,或者为几个重叠的强度范围设定几个不同的模式。这些都只取决于设计者的决定。

但在做如上所述的事情之前,必须知道如何在代码中改变播放强度级别编辑器的值。值得高兴,这同设置速度一模一样! 下面给出了获取或设置这个编辑器的代码:

```
void CAudioManager::SetPerformanceGrooveLevelModifier( int value)
```

```
{
 if (value > 100) value = 100;
 if (value < 0) value = 0;
 char v = value;
 if (NULL == m_Performance) return;
 ThrowIfFailed( m_Performance -> SetGlobalParam ( GUID_PerfMasterGrooveLevel, &v,
sizeof(v)),
    "CAudioManager::SetMasterGrooveLEvel: SetGlobalParam failed.");
}

int CAudioManager::GetPerformanceGrooveLevelModifier()
{
 if (NULL == m_Performance) return(0);
 char groove = 0;
 m_Performance -> GetGlobalParam ( GUID_PerfMasterGrooveLevel, &groove, sizeof
 (groove));
 return(groove);
}
```

这两个方法与 SetMasterTempoScaleFactor 和 GetMasterTempoScaleFactor 几乎差不多一样,区别仅仅在于这两个方法的操作对象是 GUID_PerfMasterGrooveLevel,而不是 GUID_PerfMasterTempo。

11.3.5 播放乐旨

当玩家获得一次实力增加或者漂亮地完成一件任务时,你也许希望播放一段称之为乐旨的音效。由于音频管理器已让 CDynamicMusic 类收集了所有乐旨的接口,并将其按照名字映射,播放乐旨的代码事实上只有如下几行:

```
void CDynamicMusic::PlayMotif(std::string name, bool bUseDefaultBoundary,
     bool bUseGridBoundary, bool bUseMeasureBoundary, bool bUseBeatBoundary)
{
 if (m_MotifMap.find(name) != m_MotifMap.end()) {
   CDirectMusicSegment * motifseg = m_MotifMap[name];
   int segflags = 0;
   if (bUseDefaultBoundary) segflags | = DMUS_SEGF_DEFAULT;
```

```
        if (bUseGridBoundary) segflags |= DMUS_SEGF_GRID;
        if (bUseMeasureBoundary) segflags |= DMUS_SEGF_BEAT;
        if (bUseBeatBoundary) segflags |= DMUS_SEGF_MEASURE;
         HRESULT hr = _Manager -> GetPerformance ( ) -> PlaySegment ( motifseg ->
              GetSegment( ), segflags, 0, NULL);
        ThrowIfFailed(hr, "CDynamicMusic::PlayMotif: PlaySegment failed.");
        }
    }
```

这段代码把要播放的乐旨名及指示何时播放的几个标志作为输入值(在 1/4 拍、节拍或是小节边界以及默认边界处),这些标志都有默认值以指示代码使用默认边界。大多数情况下,就使用默认边界就可以了。

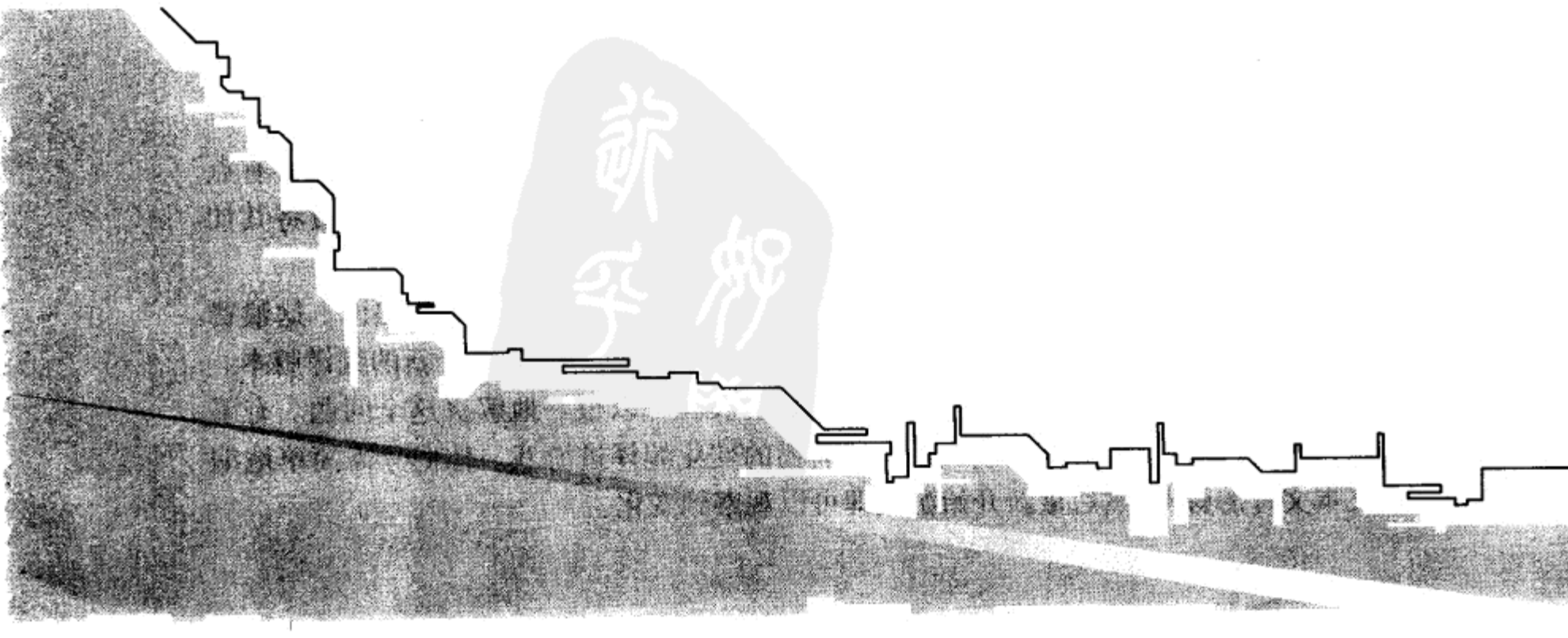
11.4 本章小结

DirectMusic 是个强大的系统,而掌握这些工具的代价是要学习一整套的新的术语和概念。读者可以根据所需功能的复杂性程度,由简入难,先创建一段仅有少量组件构成的动态音乐,如每次播放几乎相同的音乐。然后,慢慢地给音乐及代码添砖加瓦,直到完全弄明白这些对象是如何协同工作为止。

PDG

第 12 章

脚本和特效



前面已经学习了非常困难的部分，特别是动态音乐，现在读者们可以学习一些更高级的技术了。

本章讲述将两种高级技术，即 DirectX 的音频脚本技术及实时特效技术。

12.1 音频脚本

从 DirectX8.0 开始提供的音频脚本技术向游戏开发团队提供了一种并行开发过程的良好方法。初看起来，使用一种脚本语言来播放音乐好像华而不实，但在实际游戏开发团队中，它非常有用。因为它可以让音效设计者减少对程序员的依赖，并提高工作效率。

12.1.1 为什么使用音频脚本

在没有音频脚本之前，音效的设计者很难独立地工作。想象一下这样的场景：你是一个音效设计者，在一个由其他几个音效设计者和程序员（艺术家、动画设计者、制作人，和其他类型的人）组成的团队里工作，现在的工作内容是要确认魔鬼叫声在游戏中的应用。

接下来应用你奇妙的思维去思考什么样的 WAV 文件适合于表现魔鬼的叫声。你在数据库里寻找，然后发给程序员一个便条，告诉他把你选择的音效放入游戏中。程序员做完这件事后，会提供给你游戏的新的编译版本，然后你可以听到音效。

你在游戏中听到声音，但却发现并不如意。你选择的魔鬼叫声在 WAV 编辑器中听起来效果很好，但在游戏中却过于轻柔，几乎听不到。接着你发一个 mail 给程序员请他提高魔鬼叫声的音量。程序员按照你的指示做完后，再次发给你一个新的编译版本。

这次你发现这个叫声很适合蓝色魔鬼，但如果红色魔鬼也发出同样的叫声则显得有点傻。于是你又创建了一个新的 blue_orc_grunt.wav 文件，并写一个 mail 告知程序员将其加入游戏中。

你注意到发生了什么事吗？每次你想改变音频时，你必须让程序员重新编译。这浪费每一个人的时间。程序员不想常被打断，你也不想常打断他们，并等待一个新的编译版本。

通过采用实时装载的音频脚本替代音效的播放代码，可以很好地解决这个问题。有了这个技术，音效设计者再也无需依赖程序员把需要的变化编译进游戏。他们只需简单地编辑一个文本文件的脚本，然后重新开始游戏即可以观察到变化。

补充说明:周期

专业游戏开发团队依赖于游戏的调试“周期”。这里所谓周期是指两次测试的时间间隔再减去花费在修复上的时间。

对于程序员而言,周期是指停止调试器,重新编译代码,再重启调试器并回到问题点所花费的时间。对艺术家或动画设计者而言,周期指他们输出新作品(纹理、模型、动画等)到作品在游戏中可视所花费的时间。

显然,周期越短,单位时间可以做的工作就越多,最终产品就越好。如果你是一个艺术家,如果要让你在几分钟时间内使得新作品在游戏中可用,那么你可能只会花费两到三个周期以判定作品是否足够好,然后就开始处理其他事务。假如周期很短,只需要输入几个命令,就能在游戏中看到你的新作品,你就会多遍查看和编辑,游戏最终的艺术效果会更好,因为你花了更多的工夫去完善它。

所以,如果你在一个团队中工作,应尽全力使每个人的周期尽可能地短。手段很多,你可以花时间创建一个 WYSIWYG(游戏音频“即听即所得”)游戏编辑器,并尽可能让更多的事务变为数据驱动。最终,游戏会更好,团队成员也会更愉快。

程序员和音效设计者仍然需就游戏所需的音效达成一致,但程序员不需要再关心音效如何被播放的细节。比如,一个 WAV 文件是否被用作多个音效,以及它该用多大音量播放,等等。程序员只需调用由音频脚本提供的子例程,音频脚本会告知 DirectMusic 该播放什么,以及如何播放。

提示

如果你是个独立程序员,或者你的音效设计者恰巧也是个程序员,并能够直接修改游戏代码,那么音频脚本将不会特别有效。然而,如果你在一个大的团队里工作,音频脚本可以帮你大忙。

12.1.2 音频脚本基础

假设你正在创建一个关于猫的游戏。玩家控制着一只英雄猫,并通过增加其实力使猫随着时间变大。

你是这款游戏的音效设计者,创建猫的音效是你的工作。你和你的程序员决定设计一个简单的音频接口。游戏代码会跟踪英雄猫的大小,并通过一个名为 CatSize 的简单变量传递此信息。游戏代码里有一个简单的函数,名为 sfxCatMeow,每当猫应该喵喵叫时,游戏代码就会调用这个函数。sfxCatMeow 函数会检查 CatSize 的值,并创建一个合适的喵喵声。理

想状态下,应该有一系列的不同喵喵音效,以免猫每次都发出相同的喵喵声。

这个场景是专门设计的,用于说明音频脚本的有效性。有了音频脚本,音效设计者可以按照他们的喜爱,创建任意多种不同的喵喵声,同时不必操心用何种方式去组织它们,更不用修改游戏的代码。同样,程序员也不需要操心让猫喵喵叫的细节,他们只需在代码中调用一个函数,然后正确的喵喵声就会从扬声器传出来,如同耍魔术一样。

提示

拥有合适的音频接口非常重要,即音频脚本中使用的例程表和变量。如果在游戏代码和音频之间存在一个定义良好的接口,音频脚本将会工作得非常棒。理想状态下,应该仅定义一次这个音频接口。程序员和音效设计者坐到一起,决定游戏代码应该设置哪些变量,应该调用哪些函数,然后双方便各自分头做事。音效方面的人员将会在音频脚本的层面上工作,而程序员则完成游戏代码的其他部分。

实践中,这个目标永远达不到,但它是我们为之努力的方向。音频接口的小改动通常会发生,但如果你发现经常重写整个音频接口的话,那么你多半已失去了音频脚本的巨大威力。

接下来的两节将分别从音效设计者和程序员的视角讲述音频脚本是如何应用的。这段音频脚本将只有一个函数和一个变量,但要记住一个真正的游戏将会有成打的函数和变量。

你将首先学习如何创建脚本(即脚本本身的代码),然后学习如何将脚本集成到 C ++ 游戏代码中。

12.1.3 音效设计者的视角

DirectX 使用的音频脚本语言基于 Visual Basic,所以如果使用过 VB 的其他变种(Active Server Pages,Office 宏,等等),那么你多半明白你所需要做的大部分内容。如果以前从未接触过 VB,或许需要阅读一些在线的简介以了解其基本控制流和语法(请查看 CD 里提供的部分链接)。

提示

AudioVB 脚本的功能不仅限于播放音效(虽然这里只会学到这一点)。你也可以用它去播放常规的段、改变强度水平(groove level)、播放乐旨——基本上所有能在 DirectMusic Producer 里操作的东西,都可以用音频脚本操作。

可以用 DirectMusic Producer 创建 AudioVB 脚本,但事情要比创建一个脚本对象并在编

辑器中输入脚本复杂。这是因为你不能直接从 AudioVB 脚本播放一个 WAV 文件,而只能播放一个音效段。所以,必须通过脚本为每一个希望播放的音效创建一个 WAV 段。

可以把 WAV 段看作是一个包含 WAV 文件而非模式的音效段。在本章中,WAV 段不会有强度级别和风格,也不会播放模式。相反,它们会通过一个 WAV 轨迹发出声音。我把包含 WAV 轨迹的段看作是 DirectMusic 的一种快捷方式。当然,你也可以创建一个风格以及一系列的模式,创建一个风格轨迹和强度轨迹,并通过那种方式播放你的音效。对于通过简单的 WAV 元素来创建复杂的音效,这种方法是有用的。但如果仅仅想播放一个简单的音效,则大可不必那样做,可以简单地把一个 WAV 文件嵌入到段里。

(1)创建音效段

以下是必须要了解的。运行 DirectMusic Producer 并创建一个新的项目 CatProject,需要 1 个音频脚本和 2 个音效段,脚本命名为 catsfx. spp,段命名为 BigCat. sgp 和 SmallCat. sgp。

接下来,右击 CatProject 树形视图的根结点,选择 Insert Files,插入 4 个喵喵 WAV 文件(可以在本章目录中找到这 4 个 WAV 文件,其中 2 个是小猫的喵喵声,2 个是大猫的喵喵声)。浏览并选择这 4 个 WAV 文件,接下来将看到 4 个 WVP 对象。

现在需要把 4 个 WAV 文件插入到 2 个段里,每个段具有 2 个变奏(WAV 轨迹,同模式乐部一样,可以有最多 32 个变奏)。右击 BigCat 段的时间刻度轨迹(time signature track),从弹出菜单中选择 Add Tracks 以增加一个 Wave Track。

> **提示**
>
> 不需要点击 29 次变奏按钮使每个变奏可用或不可用。相反,你可以点击所有变奏按钮左边的窄按钮改变整个变奏集的状态(如图 12.1 所示)。

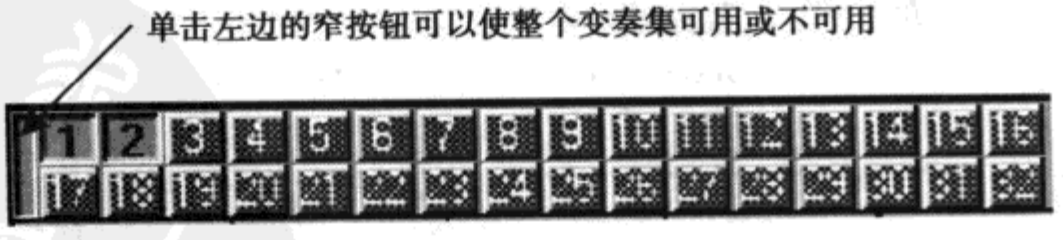


图 12.1 单击左边的窄按钮可以使整个变奏集可用或不可用

这个段看起来更像模式了。请注意新增的 Wave 轨迹(也可称为 Wave 乐部)就如模式一样具有变奏按钮。除了变奏 1 外,让其他所有变奏不可用,然后把光标移到轨迹的开始处,并点击 Insert。此时波形属性窗口将会出现,如图 12.2 所示。从中选择 CatProject: BitCat1 以告知 DirectMusic 它应为变奏 1 播放这个 WAV 文件(注意只有作为项目一部分的

WAV 文件才会出现在列表中,这就是为什么必须提前导入它们的原因)。如果一切正常,应该可以看到在轨迹的开始处,出现了一段波形图以表示那个 WAV 文件。

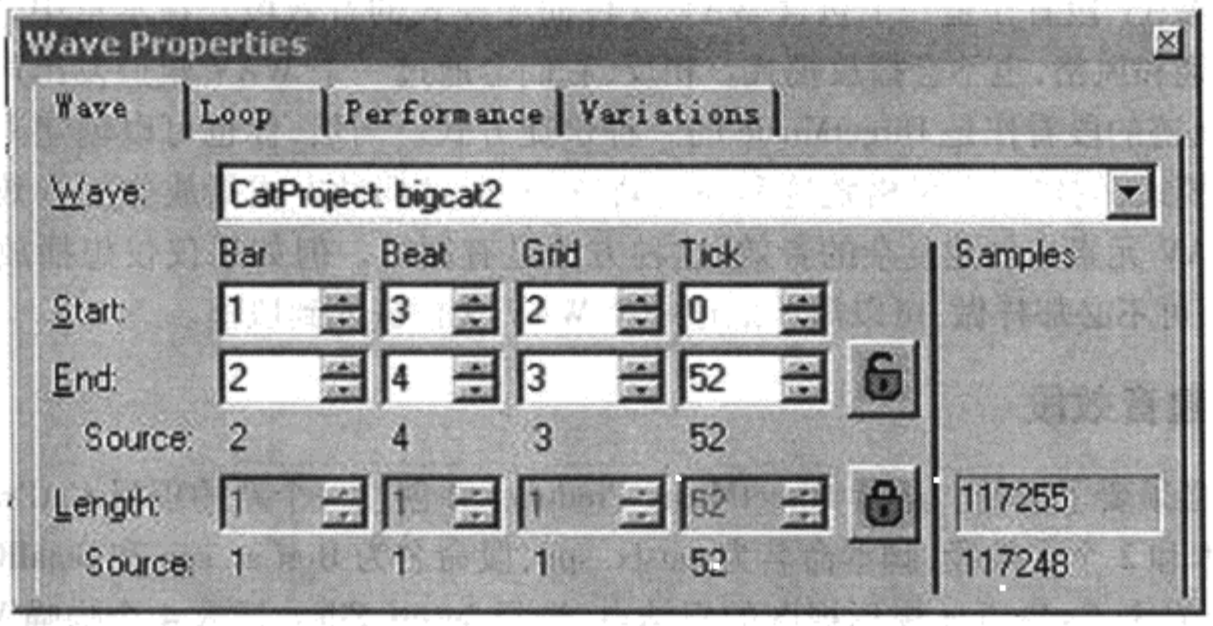


图 12.2 Wave 属性窗口

按照相同的步骤指定 bigcat2. wav 文件从变奏 2 的起始处播放。记住此时要让变奏 1 不可用,以避免意外地把 bitcat2. wav 文件同时指定给了变奏 1 及变奏 2。可以通过如下方式检查结果是否正确:通过点击变奏按钮 1 可以看到 bitcat1. wav 的波形图随点击出现或消失,而点击变奏 2 按钮则可以看到 bitcat2. wav 的变化。

还有一件重要事情就是在段窗口中,右击紫色背景的 Wave Track 名字,并选择属性。此时,DirectMusic Producer 将弹出 Wave Track 属性窗口,如图 12.3 所示。选择 Wave Track Tab 页并确认 Reset Variation order on Play(播放时重置变奏)项被选中。如果要创建一个随机播放 bigcat1. wav 和 bigcat2. wav 的段,这是必须的。

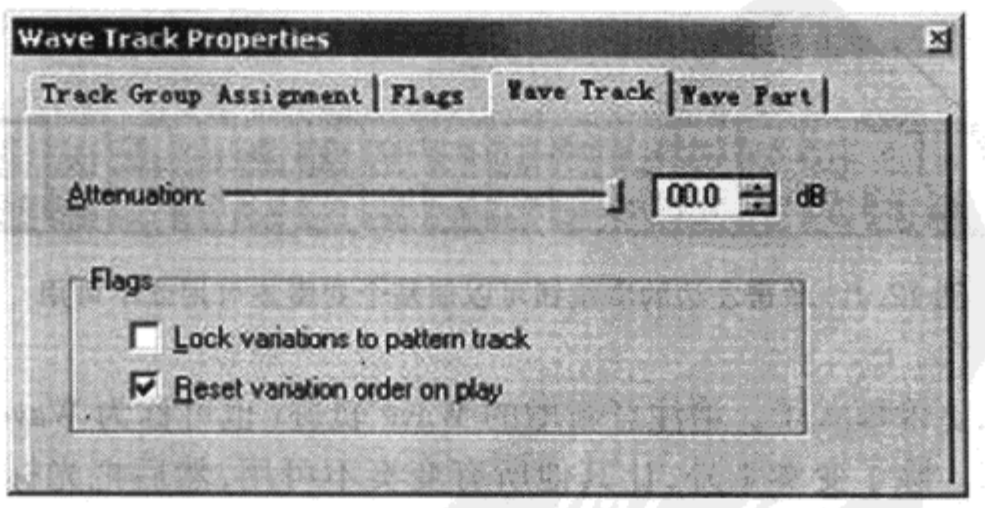


图 12.3 Wave Tracks 属性窗口

如果现在通过工具栏播放这个段,将随机地听到 bitcat1. wav 或 bitcat2. wav。如果听到两者同时在播放,说明你在编辑一个变奏时,另一个变奏正处于可用状态。图 12.4 显示了最终的段(两个变奏都是活动的)。

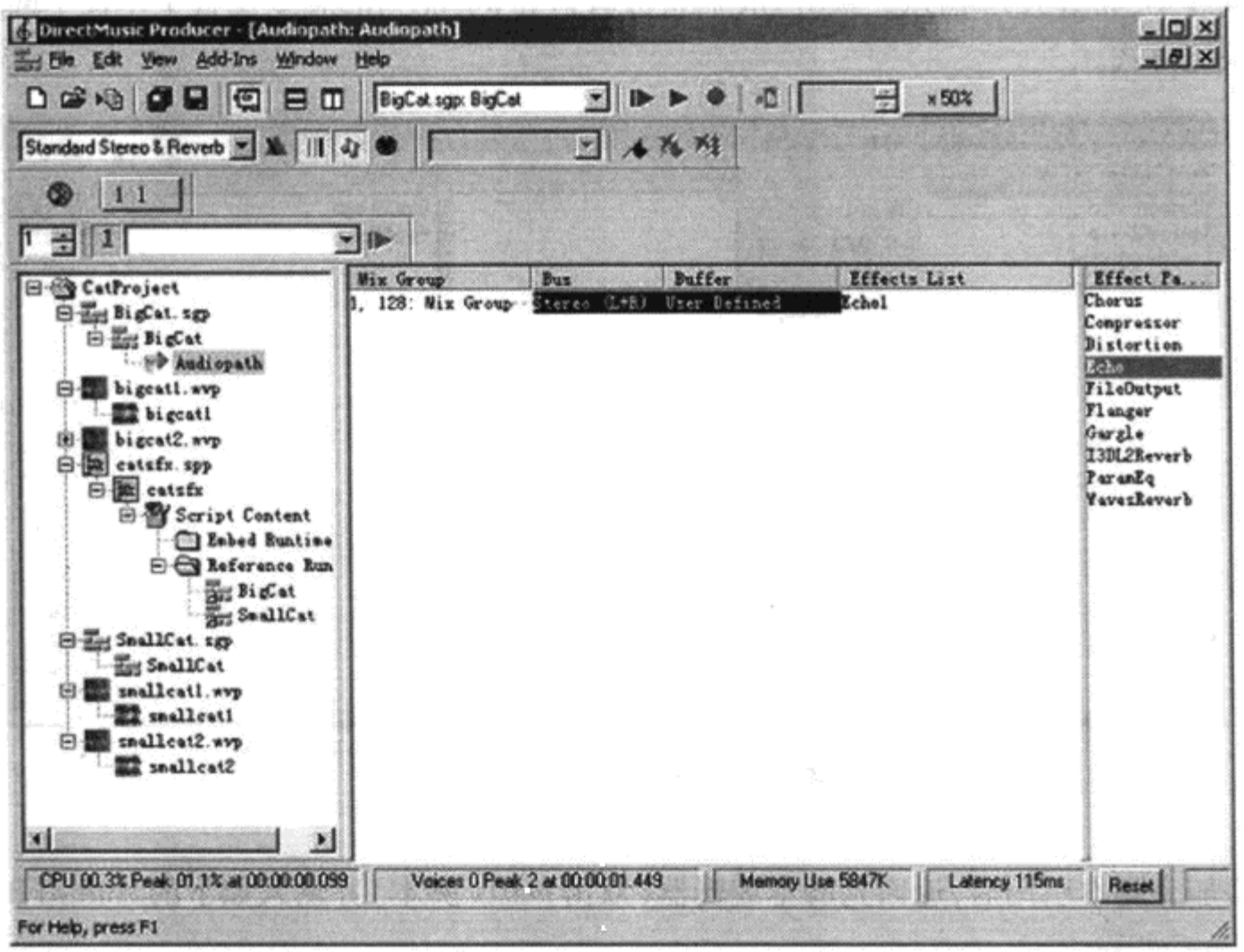


图 12.4 最终的 BigCat segment,两个变奏都是活动的

用同样的方法创建 SmallCat 段,它随机播放 smallcat1. wav 或 smallcat2. wav。同时,注意变奏是否设置正确,并用工具栏播放段以验证其正确性。

(2)创建一段简单的脚本

创建好段之后,就可以创建脚本本身了。单击工具栏上的 New 按钮并从弹出的对话框中选择 Script,给新脚本重新命名为 catsfx。

> **警告**
>
> 把对象加入音频脚本的引用中非常重要。如果不这么做,脚本将无法正常工作。

在开始书写音频脚本之前,需要告知 DirectMusic 脚本引用项目中的哪些对象。你将要编辑的脚本会用到 BitCat 及 SmallCat 两个段,需要将它们加入脚本的引用中。打开树形视图脚本下方的 Script Content 节点,然后右击 Reference Runtime,将 BigCat. sgp 及 SmallCat. sgp 加入引用列表中,如图 12.5 所示。单击 OK 后可以看到 Reference Runtime 文件下出现了两个指向段的快捷方式。

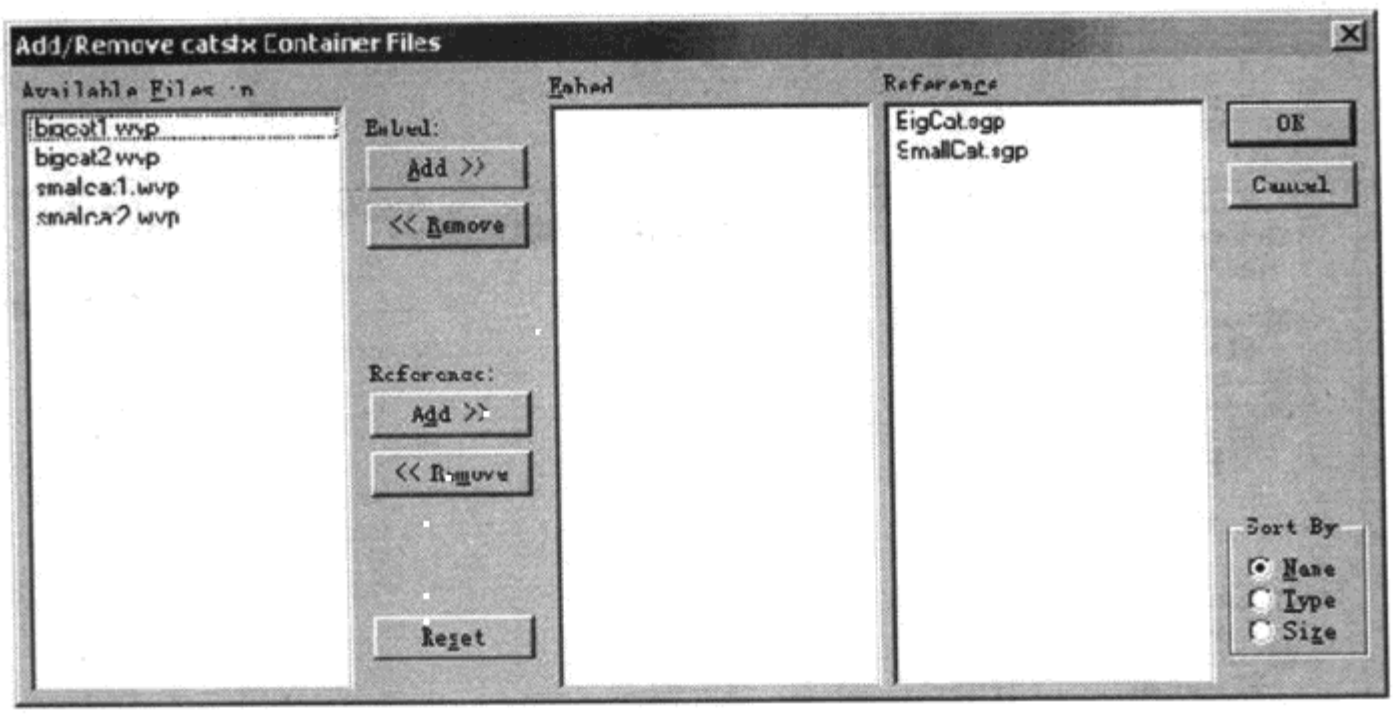


图 12.5 给音频脚本添加引用

添加完脚本引用后,返回到脚本编辑器(双击 catsfx 脚本)并输入如下的 AudioVB 代码:

```
dim CatSize

sub sfxCatMeow
    if CatSize < 10 then
        SmallCat. Play( IsSecondary)
    else
        BigCat. Play( IsSecondary)
    end if
end sub
```

注意,AudioVB 与 C ++ 语法不同。比如,不能在 AudioVB 脚本的行尾添加分号,那样会把事情弄糟。

这段脚本非常简单,它声明了一个名为 CatSize 的变量和一个名为 sfxCatMeow 的子例程。该子例程将观察 CatSize 变量的值(值由游戏设定),如果小于 10,播放 SmallCat 段,否

则播放 BigCat 段。这里，需要注意 IsSecondary 标志，它告知 DirectMusic，这个段将会作为次段播放（通常都是将音效作为次段播放，以免错误地夺取了正在播放中的动态背景音乐的控制权）。

> **提示**
>
> 编辑脚本时，可以借助 DirectMusic Producer 检查语法错误。单击编辑器右端，例程列表上方的 Refresh 按钮，所有例程的名字都会出现在例程列表中。相反，当你单击 Refresh 按钮时如果变量和例程名称没有被正确解析出来，请检查语法。

保存脚本。保存项目并将所有东西保存为运行时格式。把运行时文件拷入第 12 章示例程序的工作目录 Ch12p1_AudioScripts，可以直接把文件保存到这个目录，也可以从根目录中将运行时文件拷贝过来。或者，也可以书写代码指示 DirectMusic 搜索不同的路径，默认情况下仅搜索当前路径，如果是通过 DevStudio 运行示例程序的话，当前路径是 Ch12p1_AudioScripts。

显然，真正游戏的脚本要复杂得多。一个真正的游戏会为每个音效设计一个函数，此外，还会有控制背景音乐的函数，播放乐旨的函数，以及完成所有其他游戏所需功能的函数。如何组织这些子例程完全取决于你。可以根据需要创建任意数量的脚本文件、变量和子例程，但好的方法是为游戏的每一个级别设定一个脚本文件。

> **提示**
>
> 注意音频脚本中子例程的名字以 sfx 开头。一个好的习惯是给音频脚本规定一个某种形式的命名规范——比如，要求所有与播放音效有关的子例程都以 sfx 开头。创建（并遵守）标准命名规范可以使大段的脚本更清晰易懂。

12.1.4 开发者的视角

创建好音效段和脚本，并将其保存为运行时格式之后，就可以开始书写操作音频脚本的游戏代码。

DirectX Audio 提供了一个非常简单的且 100% 完整的音频脚本接口。在初始化音频脚本接口后，即可以要求接口提供子例程和变量的列表，也可以设定或获取变量的值，或调用子例程。这些即是你需要的全部功能，而且在大部分时间你并不需要接口提供子例程或变量的列表，因为将这些子例程或变量硬编码就已经足够了。

现在需要简单地修改一下代码。接下来将介绍如何为音频引擎添加音频脚本支持。

在 CAudioManager 的几个新对象方法的支持下，大多数的支持都会通过一个新的名为 CAudioScript 的类提供。在创建 CAudioScript 或修改 CAudioManager 之前，需要创建一个新的智能指针。

(1)一个新的、更聪明的指针类

到目前为止，你只有一个智能指针 CSoundPtr。那是因为，到目前为止所处理的对象，在基本层面都可以播放、停止和设置音量。但是你却无法像播放一个段或者 WAV 文件一样播放一段音频脚本。所以，从 CSound 继承新的 CAudioScript 是没有意义的。CAudioScript 不从任何类继承，因为它与任何现有的实体都不相似。这意味着，当要求 CAudioManager 装载一段音频脚本时，它并不会返回一个 CSoundPtr。但事实是必须返回一个指针。可以重写装载例程并返回一个 CAudioScript 指针，也可以很容易地创建一个专为 CAudioScript 设计的智能指针(从 CRefCountPtr 模板继承)。下面是指针类的代码：

```
// 指向 CAudioScript 的智能指针
class CAudioScriptPtr : public CRefCountPtr < CAudioScript >
{
public:
  explicit CAudioScriptPtr( CAudioScript * p = NULL);
  void Release( );
};
```

重量提升(the heavy lifting)由 CRefCountPtr < CAudioScript > 完成。所有的 CAudioScriptPtr 都需要一个显式的构造函数和一个 Release 函数。这两个函数的代码如下：

```
CAudioScriptPtr::CAudioScriptPtr( CAudioScript * p) : CRefCountPtr < CAudioScript > (p)
{
  stringstream str;
  if (p) {
    str << " CAudioScriptPtr:: CAudioScriptPtr: constructed for pointer " << p <<
      " name: " << p -> GetName( ) << " (count = " << itsCounter -> count <<
      ")..." << endl;
    OutputDebugString( str. str( ). c_str( ));
  }
}

void CAudioScriptPtr::Release( )
```

```
{
  stringstream str;
  if (itsCounter) {
    str << "CAudioScriptPtr::Release(): pointer " << itsCounter->ptr << " name: "
        << itsCounter->ptr->GetName() << " (count = " << itsCounter->count
        << ")..." << endl;
    OutputDebugString(str.str().c_str());
  }
}
```

这两个函数都只是输出了一些东西。构造函数将状态信息输出到 DevStudio 的跟踪输出,这样就可以清楚地看到智能指针何时被创建;Release 函数也输出一行信息方便确定内存被正确地释放。注意,这些函数输出 CAudioScript 类的名字,这个名字是由 GetName 函数获取的,后面我们将会创建 GetName 函数。同时也要注意,Release 函数事实上什么也没做,真正的指针删除是由 CRefCountPtr 完成的。

智能指针 CaudioScript 与智能指针 CSound 类似,区别仅在于不同的跟踪输出和 CRefCountPtr 的不同模板参数。由于 CAudioScriptPtr 可以显示对应的音频脚本的名称,因此可以说它比 CSoundPtr 聪明些。由于二者都会在正确的时间删除指针,它们都很聪明。

(2) CAudioScript

创建好 CAudioScriptPtr 智能指针后,现在可以创建 CaudioScript 对象类了。下面是它的声明:

```
class CAudioScript
{
public:
  friend class CAudioManager;
  CAudioScript(CAudioManager * mgr);
  virtual ~CAudioScript();
  void SetVariable(std::string varname, int value);
  int GetVariable(std::string varname);
  void CallRoutine(std::string procname);
  std::vector<std::string> EnumVariables();
  std::vector<std::string> EnumRoutines();
  std::string GetName() { return(m_Name); }
```

```
    std::string AssembleErrorInfoString(std::string header,
        DMUS_SCRIPT_ERRORINFO &errInfo);
protected:
    std::string m_Name;
    void Init(std::string name); // 保证只有 CAudioManger 可以对其进行初始化。
    CAudioManager * m_Manager;
    IDirectMusicScript8 * m_Script;
    unsigned char * m_OriginalData;
};
```

相信读者已能够理解这段声明的绝大部分内容,只有少数东西需要解释。注意,同平常一样,CAudioScript 有一个指针 m_OriginalData,这与 CSound 相同。当一个音频脚本从内存中创建时,音频管理器会设定这个指针的值。同所有的 CSound 后代一样,CAudioScript 会记住它所创建的地址,并在销毁时自动释放那块内存。同时也要注意,同 CSound 一样,CAudioScript 是 CAudioManager 的友类,所以当装载一个新的音频脚本时,CAudioManager 可以直接访问 CAudioScript 的私有和保护成员。并且当构造一个 CAudioScript 对象时,必须提供一个 CAudioManager 指针,这样每个 CAudioScript 都有一个指向 CAudioManager 实例及一个 DirectMusic 执行接口的指针。

CAudioScript 的核心是 m_Script,它是由 DirectMusic Audio 提供的 IDirectMusicScript8 接口。读者也可以看到音频脚本有一个名为 m_Name 的变量,此变量的值可以通过函数 GetName 及 SetName 获取和设定。对于游戏程序员来说,在音频脚本对象被创建或释放时,智能指针 CAudioScript 会通过跟踪输出而输出音频脚本的名字。在下一节中将会看到代码是如何决定音频脚本对象名称的。

CAudioScript 还有一个私有的 Init 函数,它确保脚本只执行一次初始化。当新脚本被装载时,CAudioManager(作为友类,可以调用保护成员)会调用这个 Init 函数。除此之外,还有很多方法,用于设定或获取变量值,调用一个子例程,枚举所有可用的变量和例程,组合错误信息等等,这些内容将在后面的小节中学到。

(3)装载并初始化脚本

在音频脚本可用之前,必须先装载并初始化它。这意味着在 CAudioManager 中需要 3 个新函数:一个用于从磁盘装载,一个从资源装载,另一个从内存中装载。下面是磁盘装载函数:

```
CAudioScriptPtr CAudioManager::LoadScript(std::string filename, std::string name)
{
```

```
CAudioScript * script = new CAudioScript(this);
HRESULT hr;

// 将 filename 转换到宽字符串
WCHAR widefilename[MAX_PATH];
DXUtil_ConvertGenericStringToWideCch( widefilename, filename.c_str(), MAX_
    PATH);

// 让装载器装载这个文件
hr = m_Loader -> LoadObjectFromFile(
   CLSID_DirectMusicScript,
   IID_IDirectMusicScript8,
   widefilename,
   (void **) &script -> m_Script);
ThrowIfFailed(hr, "CAudioManager::LoadScript(std::string filename):
   LoadObjectFromFile failed.");

  if (!name.length()) { name = filename; }
  script -> Init(name);

  return(CAudioScriptPtr(script));
}
```

现在,应该知道 LoadScript 的所有动作了。它先创建一个 CAudioScript,把指定文件名转换到宽字符串,然后命令 DirectMusic 去装载这个文件(通过指定 CLSID_DirectMusicScript 类和 IID_IDirectMusicScript8 接口),接着把 DirectMusic 提供的接口值赋给脚本对象的 m_Script 变量。完成上述工作之后,它会设定脚本对象的 Name 值并调用 Init 函数进行初始化。当 Init 函数执行完成后,它会为新脚本返回一个新的智能指针。

这里有点神秘的是 CAudioScript 的 Init 函数。以下是 Init 函数的程序代码:

```
void CAudioScript::Init(string name)
{
  m_Name = name;
  DMUS_SCRIPT_ERRORINFO errInfo;
  HRESULT      hr;
```

```
    errInfo.dwSize = sizeof(DMUS_SCRIPT_ERRORINFO);
    hr = m_Script->Init( m_Manager->GetPerformance(), &errInfo);
    if (FAILED(hr)) {
      ThrowIfFailed(hr, AssembleErrorInfoString(
            "CAudioScript::Init: m_Script->Init() failed!", errInfo));
    }
}
```

Init 函数简短而轻松。它真正完成的工作是调用音频脚本接口的 Init 函数。在使用一个脚本接口之前,必须将其初始化并告知其与哪个演奏通道相关联,这是 DirectX Audio 的要求。

注意,DMUS_SCRIPT_ERRORINFO 对象的结构变量 errInfo。这个结构的地址作为第二个参数传递给 IDirectMusicScript8::Init,如果运行出错,代码将其传递给 AssembleErrorInfoString 函数。

DMUS_SCRIPT_ERRORINFO 对象很有用。初始化 IDirectMusicScript8 接口的过程就是编译 AudioVB 代码的过程,如果代码有错,初始化过程将失败并将各种有用的信息填入 errInfo,你将获得错误的文件名、行号、错误的描述信息,以及是哪一个组件产生了这个错误,即处理脚本时出现问题的 DirectX 组件的名称。

AssembleErrorInfoString 使用指定的 DMUS_SCRIPT_ERRORINFO 对象来创建详细的错误描述信息。这个错误信息将通过 ThrowIfFailed 宏弹出,最后显示为一个类似于图 12.6 的对话框。

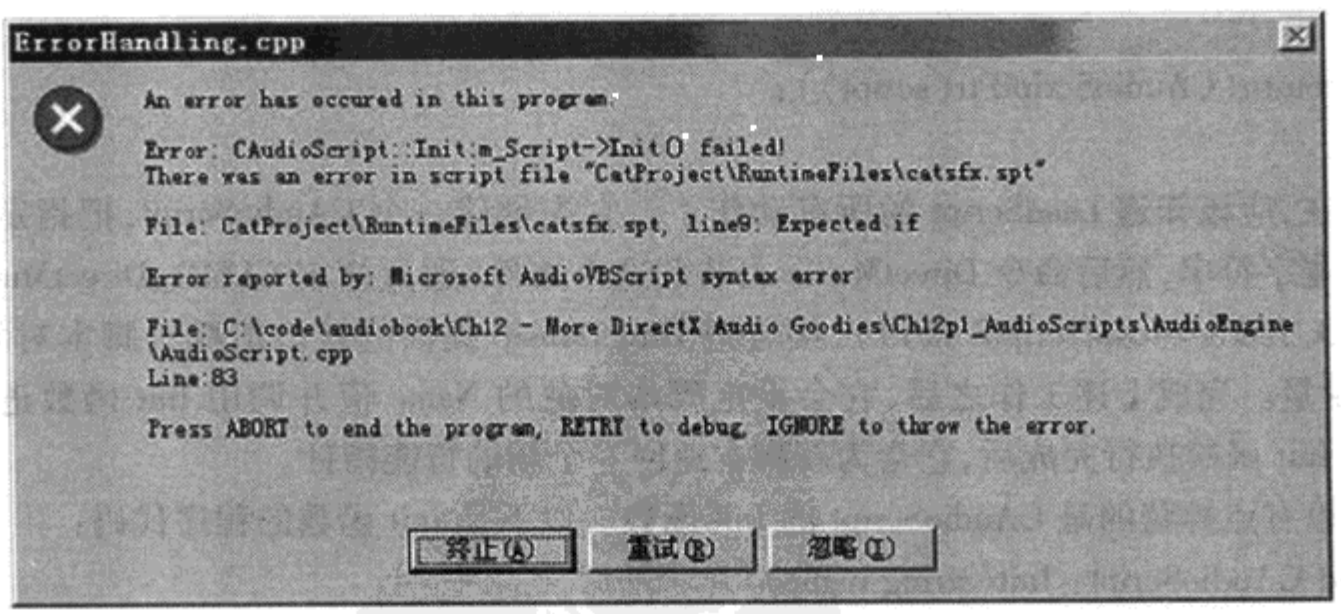


图 12.6　由 AssembleErrorInfoString 触发的错误对话框

下面是 AssembleErorInfoStirng 的代码:

```
string CAudioScript::AssembleErrorInfoString(string header, DMUS_SCRIPT_ERRORINFO
&errInfo)
{
    stringstream str;
    str << header << endl;

    // 将错误信息放入字符串中
    char file[DMUS_MAX_FILENAME];
    char desc[DMUS_MAX_FILENAME];
    char component[DMUS_MAX_FILENAME];

    DXUtil_ConvertWideStringToGeneric(file,
        errInfo.wszSourceFile, sizeof(file));
    DXUtil_ConvertWideStringToGeneric(desc,
        errInfo.wszDescription, sizeof(desc));
    DXUtil_ConvertWideStringToGeneric(component,
        errInfo.wszSourceComponent, sizeof(component));

    str << "There was an error in script file \"" << GetName() << "\"." << endl << endl;
    str << "File: " << file << ", line " << errInfo.ulLineNumber << ": " << desc << endl << endl;
    str << "Error reported by: " << component << endl;

    return(str.str());
}
```

惟一麻烦的是 DMUS_SCRIPT_ERRORINFO 的所有字符串成员都是宽字符,所以必须把它们转换成通常的 ASCII 字符串(通常是通过 DXUtil_ConvertWideStringToGeneric 函数)。完成后,便获得关于错误的详细信息,AssembleErrorInfoString 函数将所有信息输出到一个大的 STL 字符串,并返回。

(4)枚举变量和例程

装载并初始化一个音频脚本之后,可以通过枚举其变量和例程来查看其内部状况。以下是通过 EnumRoutineshshu 函数获取代码例程列表的过程:

```
vector < string > CAudioScript::EnumRoutines( )
{
  vector < string > routines;
  int count = 0;
  HRESULT hr;
  bool done = false;
  while ( ! done) {
    WCHAR widename[ MAX_PATH ];
    char name[ MAX_PATH ];
    hr = m_Script -> EnumRoutine( count, widename);
    ThrowIfFailed( hr, " CAudioScript::EnumRoutines( ): EnumRoutine failed. " );
    if (hr == S_FALSE) done = true;
    else {
      DXUtil_ConvertWideStringToGenericCch( name, widename, MAX_PATH);
      routines. push_back( name);
      count ++ ;
    }
  }
  return( routines);
}
```

这段代码循环地调用 IDirectMusicScript8 接口的 EnumRoutine 函数。这个函数有两个参数,一个是需要查询的例程的索引号,另一个是指向宽字符串的指针,这个宽字符串将用于保存例程的名称。如果给定的索引号过大,函数将返回 S_FALSE。

这种方法使得获取例程列表十分地容易。这段代码从索引号 0 开始,循环地调用 EnumRoutine 并检查返回值。如果返回值是 S_FALSE,停止循环,否则,将名字转换成通常的 ASCII 字符串,并将其推入向量中。

CAudioScript::EnumVariables 的实现与此类似,区别是它调用的是 EnumVariable 而非 EnumRoutine。

(5)设置变量值

到现在,已经装载并初始化了一个音频脚本,并通过枚举其变量和例程的列表了解了其内部细节。在调用音频脚本的某个例程之前,你或许会期望通过设置其变量的值与其通信。现在看看 CAudioScript::SetVariable 的代码:

```
void CAudioScript::SetVariable(string varname, int value)
{
  DMUS_SCRIPT_ERRORINFO errInfo;
  WCHAR widevarname[MAX_PATH];
  DXUtil_ConvertGenericStringToWideCch(widevarname, varname.c_str(), MAX_
    PATH);
  HRESULT hr = m_Script->SetVariableNumber(widevarname, value, &errInfo);
  if (FAILED(hr)) {
    ThrowIfFailed(hr, AssembleErrorInfoString("CAudioScript::SetVariable:
      SetVariableNumber failed!", errInfo));
  }
}
```

设置一个整型值只需要调用 IDirectMusicScript8 接口的 SetVariableNumber 函数。其参数是希望设置值的变量的名称(宽字符串)及期望设定的值。如果变量存在,DirectX Audio 会正确地设定它的值,否则,将填充一个 DMUS_SCRIPT_ERRORINFO 结构以报告错误。

提示

可以通过与设置整型值相似的过程获取或设定一个字符串值。可以查询 DirectX Audio 的文档以获取关于 SetVariableVariant 方法的信息。

(6)获取变量值

获取一个变量的当前值也非常容易,其代码如下:

```
int CAudioScript::GetVariable(string varname)
{
  DMUS_SCRIPT_ERRORINFO errInfo;
  WCHAR widevarname[MAX_PATH];
  long value=0;
  DXUtil_ConvertGenericStringToWideCch(widevarname, varname.c_str(), MAX_
    PATH);
  HRESULT hr = m_Script->GetVariableNumber(widevarname, &value, &errInfo);
  if (FAILED(hr)) {
    ThrowIfFailed(hr, AssembleErrorInfoString("CAudioScript::GetVariable:
```

```
            GetVariableNumber failed!", errInfo));
    }
    return(value);
}
```

GetVariableNumber 函数把指定宽字变量名的值放入一个长整型指针里,如果有错误发生,它也会填充一个 DMUS_SCRIPT_ERRORINFO 结构以报告错误。

(7)调用例程

设置好变量的值后,调用一个例程只需要调用 IDirectMusicScript8 接口的 CallRoutine 函数,代码如下:

```
void CAudioScript::CallRoutine(string procname)
{
    DMUS_SCRIPT_ERRORINFO errInfo;
    WCHAR wideprocname[MAX_PATH];
    long value=0;
    DXUtil_ConvertGenericStringToWideCch(wideprocname, procname.c_str(), MAX_
        PATH);
    HRESULT hr = m_Script->CallRoutine(wideprocname, &errInfo);
    if(FAILED(hr)){
        ThrowIfFailed(hr, AssembleErrorInfoString("CAudioScript::CallRoutine:
            CallRoutine failed!", errInfo));
    }
}
```

同以前的一样,不要管 errInfo 错误信息的报告过程。我们只需将过程的名字转换成宽字符串,然后调用 CallRoutine 函数,DirectX 会替我们完成其他的工作。

关于音频脚本子例程,最后有几点需要说明。

首先,它们同步地执行,在遇到 end sub 行之前,CallRoutine 不会返回,所以你的游戏代码可能会因为有缺陷的代码而陷入死循环并被挂起。

第二,你无法向音频脚本例程传递参数。例程的行为是通过全局变量来约定的。这就要求变量及例程的命名必须遵循命令规范;可以考虑把例程的名字作为变量的前缀,以区分不同例程所使用的不同参数。

最后,记住例程可以被 DirectMusic 本身调用。例如,可以把对音频脚本例程的调用嵌入其他 DirectMusic 对象中。这样做并不可怕,实际上在某些情况下,这种方法还非常有用。

(8)游戏中的音频脚本

现在已有了完整的 CAudioScript 对象,需要将它应用于实际的程序中。下面是关于如何应用 CaudioScript 来完成本章开头所描述的猫喵喵叫的场景的简单说明。

完整的代码包含在 Ch12p1_AudioScripts 示例中。显然,它比真实的游戏示例要简单得多,但它基本描述了在游戏中应用音频脚本的过程。

12.1.5 音频脚本小结

现在已经大致了解了音频脚本的工作原理,如果此前从未在游戏中集成过脚本语言,那么读者一定学到了不少新东西。想知道关于音频脚本的更多信息,请参见 DirectXAudio 帮助文档,它位于 DirectXAudio \ UnderstandingDirectXAudio \ AudioScripts 及 DirectXAudio \ UsingDirectXAudio \ UsingAudioScripts,同时,也可以浏览 CD 提供的链接。

12.2 音频特效

实时音频特效的合理应用会使游戏更加精彩。当游戏者走进一个洞穴时加入一点回声,或者在通信信道不佳时,让声音失真一点,可以让玩家增加沉醉感。

DirectX Audio 提供几种内置的特效,如回声、加入滑音、齐声、失真等(完整列表请查询位于 DirectXAudio \ UsingDirectXAudio \ UsingEffects \ UsingStandardEffects 中的 DirectX 文档)。同时,开发者也可以自定义特效并将其应用在游戏中。

有两种方法可以应用特效:可以在 DirectMusic Producer 中将其设置好或者在运行时应用它。应该运用哪种方法取决于你的游戏音频是如何设计的。

如果特效在播放过程中不改变,那么在 DirectMusic Producer 中应用它是非常简便的。例如,如果你的 DirectMusic Producer 项目是为一个全部发生在洞穴中的游戏设计的,那么可以简单地编辑默认音频路径,给所有的声音都加上回声效果。但如果玩家在洞穴门口出出进进,这个方法就不行了,必须在游戏运行时根据地形(在洞里或洞外,由当前玩家的位置决定)加入或去除回声效果。

后续的小节将描述这两种方法。

12.2.1 用 DirectMusic Producer 添加特效

在 DirectMusic Producer 中添加特效十分容易,所需要做的事情就是给需要加入特效的音效段创建一个音频路径。右击 segment 并选择 NewAudioPath。通常,你会希望你的音频路径基于一个标准路径之上,如 Stereo Sound 或 3D Sound,然后选择合适的选项(如果弄不

清这些选项的含义,那就选 Standard Stereo 好了)。图 12.7 是音频路径的编辑器。

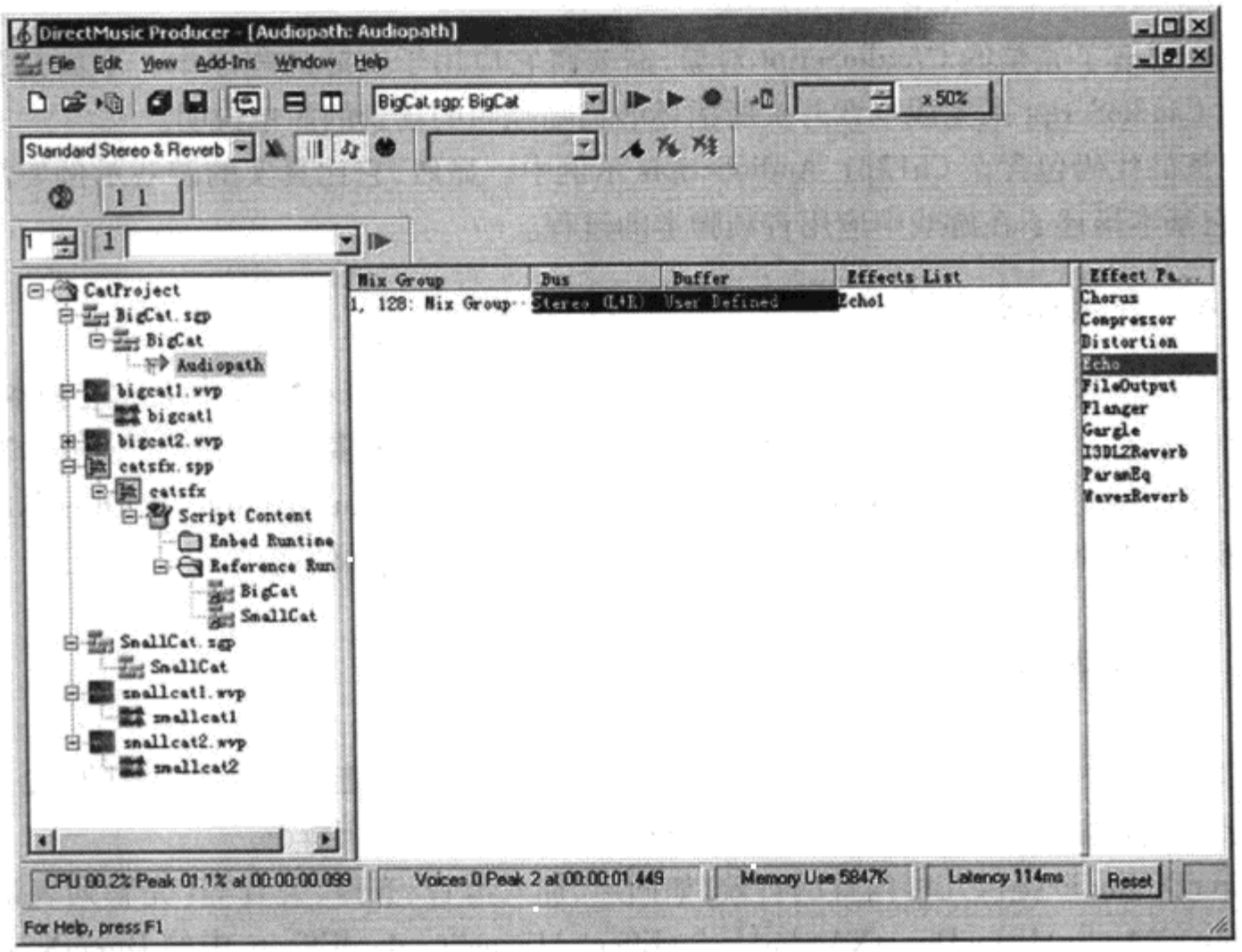


图 12.7 音频路径编辑器

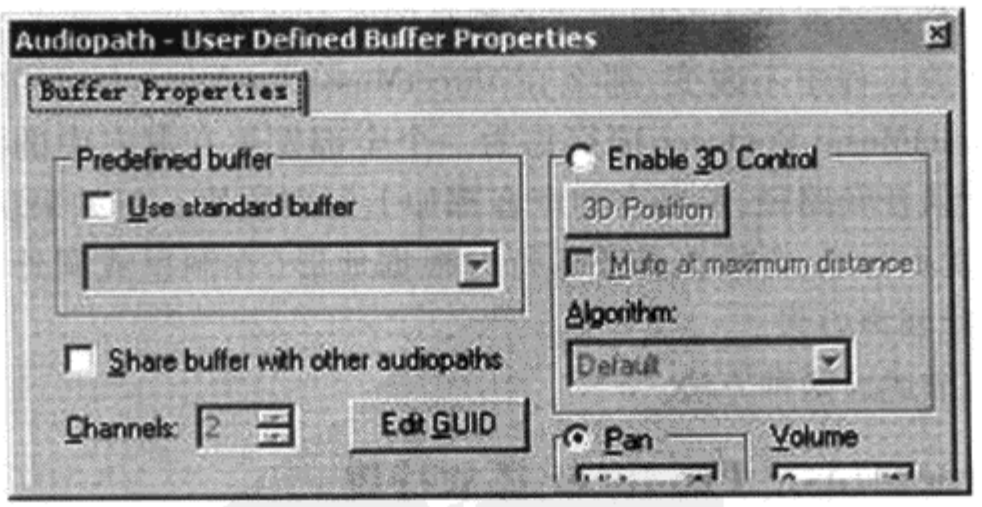


图 12.8 去除 Use Standard Buffer 复选框以给音频路径添加自定义特效

右击1～128 mix group,并选择Properties。要加入自定义特效,需要先取消Use Standard Buffer选择框,如图12.8。接着,关闭属性框,然后从编辑器右端的特效列表中将需要应用的特效拖放到对应的mix group中,就可以看到新的特效实例出现在特效列表列中。

添加完一个特效之后,右击它,选择Properties,再点击Edit Effect Properties编辑其属性。点击这个按钮后,会出现特效属性对话框,如图12.9。可以剪切、复制、粘贴特效,也可以随时播放以感觉变化。

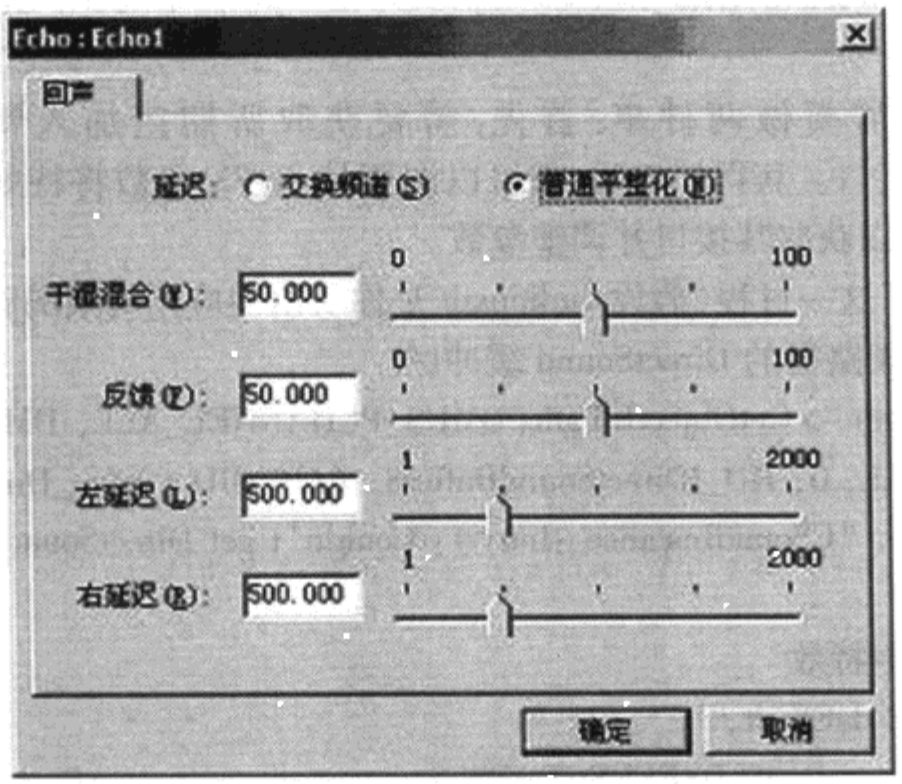


图12.9 回声的特效属性对话框

12.2.2 添加特效

在运行时添加特效比运用DirectMusic Producer要复杂一些,它将教授另外一项技术,即怎么获取一个音频路径(以DirectSound Buffer为基础)的接口。

在第2章里应用DirectSound,我们介绍了如何将原始字节直接放入DirectSound的辅缓冲区,并将其与DirectSound的主缓冲区混合,然后输出到扬声器。在DirectMusic里,同样过程也在发生,区别是由你的DirectMusic合成器而不是代码产生这些字节。

实际上每一个DirectMusic音频路径就是一个DirectSound缓冲区,而每个缓冲区都可以应用多种特效。随着这些辅缓冲区与主缓冲区的混合,DirectX Audio就可以正确地应用特效。在运行时开启或关闭一个特效意味着DirectX Audio在获取正确的DirectSound缓冲区后会插入或去除这种特效。

绑定到缓冲区的每一个特效都有自己的参数结构以及获取、设置这些参数值的接口。

例如,如果想设置一个回声的延迟,则先获取其 IDirectSoundFxEcho8 接口,设定 DSFXEcho 参数结构的适当成员,然后将这个结构传递给 IDirectSoundFXEcho8 的 SetAllParameters 函数。

(1)对 DirectX 音频特效编程

每一个特效都有一个 GUID 值,这个值定义在 dsound.h 中。比如,回声特效的 GUID 的名称是 GUID_DSFX_STANDARD_ECHO。这些 GUID 用于向 DirectX Audio 指明你所需要的特效。

使用一个特效需要做两件事:首先,需要获取所期望加入特效的音频路径的 DirectSound 缓冲区接口。获得接口后,就可以调用其 SetFX 函数将特效绑定到缓冲区。当特效被绑定后,便可以获取其接口并调整参数。

以下代码显示了这一过程(假设 audiopath 是你所期望应用特效的音频路径的接口):

```
// 取得支撑音频路径的 DirectSound 缓冲区
hr = m_AudioPath -> GetObjectInPath( DMUS_PCHANNEL_ALL, DMUS_PATH_BUFFER,
   0, GUID_NULL, 0, IID_IDirectSoundBuffer8, (LPVOID * )&m_DirectSoundBuffer);
ThrowIfFailed(hr, "CSoundInstance::Init(): Couldn't get DirectSound buffer interface.");

// 增加一个回声特效
DSEFFECTDESC dsEffect;
dsEffect.dwSize = sizeof(DSEFFECTDESC);
dsEffect.dwFlags = 0;
dsEffect.guidDSFXClass = GUID_DSFX_STANDARD_ECHO;
dsEffect.dwReserved1 = 0
dsEffect.dwReserved2 = 0;
DWORD dwResults;
hr = directsoundbuffer -> SetFX(1,&dsEffect,&dwResults);
ThrowIfFailed(hr,"Couldn't set effect.");

// 取得特效接口
hr = audiopath -> GetObjectInPath( DMUS_PCHANNEL_ALL,
      DMUS_PATH_BUFFER_DMO, 0, GUID_All_Objects, 0,
        IID_IDirectSoundFXChorus8, (LPVOID * ) &effect);
ThrowIfFailed(hr, "CChorusEffect::SetAllParameters(): couldn't get effect interface.");
```

```
// 设置特效的参数
DSFXEcho params;
params. fWetDryMix = 50;   // 50% 回声 wet),50% 原始声波(dry)
params. fFeedBack = 50;    // 50% 的输入回馈到输入
params. fLeftDelay = 750;  // 750 ms 后,回声从左扬声器发出
params. fRightDelay = 500; // 500 ms 后,回声从右扬声器发出
params. lpandelay = 0;     // 不要交换回声
hr = effect -> SetAllParameters( &params) ;

//释放特效的接口
SAFE_RELEASE( effect) ;

//释放缓冲区的接口
SAFE_RELEASE( directsoundbuffer) ;
```

这些代码不会同时出现。在现实中,它们通常分布于几个不同的地方。这些代码反应了将回声特效应用于特定音频路径(或者说特定音频路径的 DirectSound 缓冲区)的最简步骤。

这段代码首先获取音频路径的 IDirectSoundBuffer8 接口并将其存储于 directsoundbuffer 变量。接下来,代码组织一个 DSEFFECTDESC 结构,这个结构告知 DirectX Audio 将会应用何种类型的特效。缓冲区接口的 SetFX 函数允许一次性添加数个特效;它的参数包含一个 DSEFFECTDESC 结构的数组,而每一个结构都包含了指明何种特效将会被应用的 GUID 值。

SetFX 函数的第一个参数指明给予了多少个 DSEFFECTDESC 结构,第二个参数为数组中第一个结构的地址,第三个参数为一个 DWORD 数组的地址,数组的长度与 DSEFFECTDESC 数组相同。如果在应用某一特效的过程中出现错误,DirectX Audio 会将错误代码写到对应 DWORD 数组元素中。这样可以提供帮助信息以确定是什么原因导致了 SetFX 的失败。

当 SetFX 成功返回后,音频路径便已成功绑定了回声特效,但特效的参数是默认的。所以,代码将通过调用 GetObjectInPath 获取特效的接口,然后通过修改 DSFXEcho 结构的成员以调整参数,接着调用 SetAllParameters 方法将参数修改应用于特效。最后,代码释放刚才获得的接口。

再次声明,这些代码都是假想出来的。现实中它们通常分布于几个不同的类,这在后面将会学到。

(2)将特效加入音频引擎

将音频特效添加到音频引擎需要对 CSoundInstance 类作几处改动,还需要创建几个新类。通常,需要为 DirectX Audio 支持的所有标准特效各创建一个类,如加入回声、滑音、齐声等。

本节所提供的 C ++ 音频引擎代码完成后,就可以非常容易地使用特效。回忆一下第 5 章的内容,那时如果希望同时播放多个声音,就必须为每个声音创建一个 CSoundInstance 实例:

```
CSoundInstance inst[ numinsts ];
for  ( int q =0; q  <  numinsts; q ++ )  {
    snd -> Play( &inst[ q ]);
    Sleep(500);
}
```

这一段取自 Ch5p2_SoundInstances 的代码展示了一个基本思想:创建多个 CSoundInstance 结构,每次播放一个声音时,都提供给 Play 函数一个惟一的实例供其使用。

作者做了一些改进,以使得特效可以按照同样的过程被应用。应用一个特效需要做的事情是把它添加到一个声音实例中,所有采用该声音实例播放的声音都将应用那个特效。例如,下列代码所完成的工作与前段代码相同,区别在于每一个实例都绑定了回声特效:

```
CSoundInstance inst[ numinsts ];
for  ( int q =0; q  <  numinsts; q ++ )  {
    CEchoEffect  * echo  =  new CEchoEffect;
    // 为回声特效设置参数
    inst[ q ]. AddEffect( echo );
    // 一调用完 AddEffect,引擎便获得特效指针
    snd -> Play( &inst[ q ]);    // 带回声播放!
    Sleep(500);
}
```

这里显示了所需的步骤:首先,创建一个新的回声特效类 CEchoEffect。通常,当创建完这个类后,需要设定其 m_Params 变量以指定特效的参数,但这里为节约篇幅将其略去。接下来,调用 CSoundInstance 的 AddEffect 函数(后面将会讲),回声特效便神奇地绑定在那个实例上,任何通过此实例播放的声音都将带有回声特效。

提示

注意音频引擎仅允许一个特效类的一个实例，即，不可以在同一个音频路径中应用两个回声特效。我这样做的目的是希望代码简单一些；如果你希望应用同一特效的多个实例，也没有问题。

可以重复地调用 AddEffect 去添加其他特效，如果想清除所有特效，则调用 RemoveAllEffects。作者没有花时间去创建一个可以移除单个特效的方法，因为游戏通常不这样做。如果确实需要，创建一个这样的方法也很容易。

现在已经说明了从用户的角度应该怎样应用，下面讲解怎么修改音频引擎的代码以实现这些功能。

(3)特效基类 CEffect

在音频引擎中，每个不同的特效都由一个特效类来表示。所有特效类都继承于 CEffect 基础类。CEffect 类的代码如下所示：

```
class CEffect
{
public:
    CEffect();
    virtual ~CEffect();
  virtual GUID GetGuid() = 0;
  virtual void SetAllParameters(IDirectMusicAudioPath8 *audiopath) = 0;
};
```

GetGuid 及 SetAllParameters 是 CEffect 基础类的重要函数。这两个函数都是虚函数，所以它们必须被所有的特效继承类所定义。GetGuid 函数返回特定特效的 GUID 值(如 GUID_DSFX_STANDARD_ECHO)，而 SetAllParameters 函数用于为指定的音频路径上的特效设定参数。SetAllParameters 将在下一节学习。

下面是简单的回声特效类：

```
class CEchoEffect : public CEffect
{
public:
  CEchoEffect() { ZeroMemory(&m_Params, sizeof(DSFXEcho)); }
  virtual ~CEchoEffect() { }
```

```
    virtual void SetAllParameters(IDirectMusicAudioPath8 * audiopath);
    virtual GUID GetGuid() { return(GUID_DSFX_STANDARD_ECHO); }
    DSFXEcho m_Params;
};
```

可以看出,回声特效类继承于 CEffect 类,并实现其两个虚函数。为了更加简便,在这里作者把 m_Params 做为公开变量。如果需要,可以保护它,并创建合适的访问函数以获取或设置其特定变量(比如,SetWetDryMix 或 GetWetDryMix)。当然也可以在构造函数中给 m_Params 结构设定合适的参数值,而不是设定其所有的值为 0。

(4)改进 CsoundInstance 类

CsoundInstance 类包含如下几个函数和变量:

```
class CSoundInstance
{
public:
    // 有删节,以节省篇幅
    void AddEffect(CEffect * effect);
    void RemoveAllEffects();
    void ApplyEffectsToAudioPath();
protected:
    void ClearEffectsVector();
    std::vector < CEffect * > m_Effects;
    IDirectSoundBuffer8 * m_DirectSoundBuffer;
};
```

首先,成员变量 m_Effects 是一个 CEffect 指针的 STL 向量。这个向量包含了与此实例相绑定的所有特效的列表。第二,m_DirectSoundBuffer 是一个 DirectSound 缓冲区指针,这个指针通过音频路径获取,调用其 SetFX 方法可以应用特效。

在 CSoundInstance 中还有许多其他方法。公开的特效接口包含 3 个方法:AddEffect、RemoveAllEffects 及 ApplyEffectsToAudioPath。ApplyEffectsToAudioPath 方法是一个刷新方法,如果在特效添加后修改了特效的参数,需要手工的调用这个函数以将新参数应用到特效。CSoundInstance 类中还有一个保护方法,ClearEffectVector,它将会在类析构时由 RemoveAllEffects 调用,用于从 m_Effects 数组中删除所有指针。

AddEffect 方法很简单:

```
void CSoundInstance::AddEffect(CEffect * effect)
{
  m_Effects.push_back(effect);
  ApplyEffectsToAudioPath();
}
```

这段代码将新的特效指针推到向量中,然后调用 ApplyEffectsToAudioPath 函数。真正将特效应用到音频路径的工作是由 ApplyEffectsToAudioPath 完成的。

RemoveAllEffects 也比较简单:

```
void CSoundInstance::RemoveAllEffects()
{
  ClearEffectsVector();
  ApplyEffectsToAudioPath();
}
```

RemoveAllEffects 仅仅是清除特效向量,然后再调用 ApplyEffectsToAudioPath 将变化通知给 DirectX Audio。

下面是这个重要的 ApplyEffectsToAudioPath 函数:

```
void CSoundInstance::ApplyEffectsToAudioPath()
{
  HRESULT hr;
  if (NULL == m_AudioPath || NULL == m_DirectSoundBuffer) return;
  // 使音频路径失去活动状态
  m_AudioPath -> Activate(false);
  // 移除所有可能绑定的特效。
  hr = m_DirectSoundBuffer -> SetFX(0, NULL, NULL);
  ThrowIfFailed(hr,
    "CSoundInstance::ApplyEffectsToAudioPath(): Couldn't remove effects chain.");
  if (m_Effects.size() == 0) {
    m_AudioPath -> Activate(true);
    return;
  }
  // 特效向量不为空,遍历它,并将所有的特效应用到音频路径
  std::vector < DSEFFECTDESC > effectdescs;
  std::vector < DWORD > effectdescresults;
```

```
for (std::vector<CEffect * >::iterator i = m_Effects.begin(); i != m_Effects.end();
    ++i) {
  DSEFFECTDESC edesc;
  edesc.dwSize = sizeof(DSEFFECTDESC);
  edesc.dwFlags = 0;
  edesc.guidDSFXClass = (*i)->GetGuid();
  edesc.dwReserved1 = 0;
  edesc.dwReserved2 = 0;
  effectdescs.push_back(edesc);
  effectdescresults.push_back(0); // 虚构的值,以分配空间
}
hr = m_DirectSoundBuffer->SetFX(effectdescs.size(),
  &effectdescs[0], &effectdescresults[0]);
ThrowIfFailed(hr, "CSoundInstance::ApplyEffectsToAudioPath():
  creation of effects chain failed.");
// 在这里可以添加更多的详细错误信息
// (比如,检查结果向量以确定是哪一个特效失败了)
// 现在特效已创建,获取它们的接口并设置其参数
for (std::vector<CEffect * >::iterator q = m_Effects.begin(); q != m_Effects.end();
    ++q) {
  (*q)->SetAllParameters(m_AudioPath);
}
// 激活音频路径
m_AudioPath->Activate(true);
}
```

这段代码的工作是检查 m_Effects 向量的值并将这些特效应用到音频路径。首先,代码让音频路径失去活动性,因为在音频路径处于活动状态时调用 SetFX 会失败。接着,它以参数 0 及一系列的 NULL 值调用 SetFX,去除那些可能已经应用的所有特效。只有当所有的特效都已从 m_Effects 向量移除后,此操作才会真正生效。

注意,如果 m_Effects 向量为空,代码将返回。如果向量不为空,代码则遍历它并创建一个 DSEFFECTDESC 结构的向量。向量实际上就是数组,通过使用向量,可以避免正确设定数组尺寸的麻烦;对于向量,只需把元素推进去就可以了。对于 m_Effects 中的每一个特效,代码会询问其 GUID 值,并将 GUID 放入新的 DSEFFECTDESC 结构,并将结构加入到

effectdescs 向量。注意,代码也向 effectdescresults 数组推入一个 0。这样,当循环结束时,effectdescresults 便已经足够大,以保存 SetFX 的结果代码。

当循环结束后,代码就会调用 SetFX 方法,将其创建的向量及需要应用的特效的个数(即 m_Effects 向量的大小)作为参数。代码并没有对 SetFX 函数做详细的错误检查。当 SetFX 失败时,检查 effectdescresults 以确定是哪一个特效导致了 SetFX 的失败。

在 SetFX 成功返回后,代码重新遍历 m_Effects 向量中的每一个类并告知其向音频路径应用它的参数。这些都完成后,代码重新激活音频路径并返回。

现在,剩下需要解释的就是特效类如何将其参数应用到音频路径。

(5)使用特效参数

每个特效类都有一个 SetAllParameters 函数,这个函数用于将类中的参数应用到音频路径上的 DirectX Audio 特效对象。以下是 SetAllParameters 的代码:

```
void CEchoEffect::SetAllParameters(IDirectMusicAudioPath8 * audiopath)
{
  HRESULT hr;
  IDirectSoundFXEcho8 * effect = NULL;
  // 获取特效的接口
  hr = audiopath -> GetObjectInPath(DMUS_PCHANNEL_ALL,
       DMUS_PATH_BUFFER_DMO, 0, GUID_All_Objects, 0, IID_IDirectSoundFXEcho8,
       (LPVOID * ) &effect);
  ThrowIfFailed(hr, "CEchoEffect::SetAllParameters(): couldn't get effect interface.");
  // 设置特效参数
  hr = effect -> SetAllParameters(&m_Params);
  // 释放特效接口
  SAFE_RELEASE(effect);
}
```

主要工作分为 3 步:获取特效的接口,调用接口的 SetAllParameters 方法,然后释放接口。需要注意,这个接口来源于指定的音频路径,而不是 DirectSound 缓冲区。同时也需要注意,所有的特效都使用 DMUS_PATH_BUFFER_DMO 标志,它们仅仅区别于 IID。这是因为在 DirectX Audio 里,特效事实上是 DirectX Media Objects(DirectX 媒体对象,简称 DMOs)。它们被其他的 DirectX 组件(如 DirectShow)所使用,也可以直接被其他程序所使用。

每一个特效类都需要各自的 SetAllParameters 版本,因为每一个特效类都需要调用不同的 DirectX Audio 特效接口,并提供不同的 m_Params 结构。读者可以自己书写一个模板来

避免代码重复。

(6)特效编程小结

前面已经介绍了一个特定特效——回声特效的全部工作过程。CD 中所附的源代码包含了所有的 DirectX 标准特效的实现(比如 CGargleEffect、CFlangeEffect 及 CParamEQEffect)。Ch12p2_Effects 示例演示了这些特效的效果。可以编辑示例的代码以使用其他特效,或者修改特效参数以获得不同效果。在 DirectMusic Producer 中做这些工作会比较容易,因为不需要在每次修改完参数后重新编译代码。

12.2.3 合理应用特效

请注意合理地应用特效。就像辉光(lens flares)及图形暴力(graphic violence)特效一样,特效只有为特定目的有策略地应用,才会取得好的效果。通常必须很小心地应用它们。一个常犯的错误是看到这些很酷的特效就迫不及待地将它们应用于你的游戏。要避免这种倾向,有节制、精确地应用特效,游戏音效才会真正的好。

要知道关于特效的更多细节,请参见位于 DiretXAudio\UsingDirectXAudio\UsingEffects 的 DirectX 文档。

12.3 本章小结

在本章中,学习了 DirectX Audio 的两种不同技术。这些技术并不是游戏所必须的,但在某种情况下,运用它们将会非常有帮助。

第 13 章

DirectX Audio 中的 3D 音效

2D音效对你而言足够了吗？你是否还是用手工设置声音的音量和排位？你希望你的游戏有很酷的特效(如多普勒效应)吗？请继续读下去，本章将教你做这些工作，主要介绍 DirectX Audio 下的 3D 音效的概念及技术。

13.1 3D 音效的概念

开始学习如何为 3D 音效编码之前，应该弄清楚什么是 3D 音效，一个游戏具有 3D 音效指的是什么。对于一个游戏，不论从图形还是声音来讲，判断它是否是一个 3D 游戏都有不同的评估等级。比如，游戏“第一射手”(first-person shooters)有完全的 3D 图形(极可能也有完全的 3D 音效)，但游戏 RTS 呢？它的背景是 2D 的，物体却是 3D 的。

要理解什么是 3D 音效，必须先了解人类大脑是如何理解声音的。研究声音感知的科学家发现，人类大脑根据如下信息来确定声音来自相对于听者的哪个方向：

- 音量(loudness)：音量越大，大脑认为声音越近。当然，不同的声源具有不同的自然音量，但对于一个固定的声音样本，音量越大，大脑认为它越近。
- 双耳强度差异(interaural intensity difference)：一个在听者左面发出的声音当然也会窜进右耳，只是音量小一些。这将提示大脑，声音来自左边。
- 双耳时间差异(interaural time difference)：与上一概念相似，从左面发出的声音到达右耳的时间会比左耳延迟 1 毫秒左右。
- 抑止(muffling)：耳朵形状最适于接收来自于对面的声音。如果一个声音来自背后，它将被部分抑止。要扩大这种效应，将手置于耳后，手掌向前，然后分别倾听来自前方和后面的声音，就将会发现二者有所区别。神奇的是，科学家可以用数学方法模拟这种差异。

幸运的是，DirectX Audio 能帮你处理所有细节，你需要做的事情就是告诉它声音来自虚拟世界的何方，以及玩家的耳朵在哪里。

13.1.1 听者和缓冲区

在 DirectX Audio 的 3D 世界里，只有两种类型的对象：听者(Listeners)和缓冲区(Buffers)。缓冲区创建声音，而听者倾听声音。做一个比喻，在 2D 游戏 THX 里，听众是一堆人，而在 3D 世界里，听众是一个听者。通常会有几个缓冲区和一个听者。要计算一个特

定的缓冲区从扬声器中发出的声音，DirectX 需要考察缓冲区和听者之间的相对位置。

提示

如果一棵树在森林里倒下，而附近没有人，还会有声音吗？如果一个包含“倒树”样本的 3D 缓冲器开始播放，而同时不存在 3D 听者，则什么声音都不会有。在 3D 音效环境，要想听到扬声器发出的声音，至少需要一个缓冲器和一个听者。

例如，如果听者在 3D 世界的(0,0,0)位置，而有一个链锯声缓冲区位于 3D 世界的(-10,0,0)位置，那么声音将会大部分从左扬声器发出。如果链锯声缓冲区位于(-20,0,0)位置，则声音的大部分依然来自左扬声器，只是音量已经降低了。

提示

再说一次，如果一个 3D 音效缓冲区位于听者的左边，这并不意味着声音只从左扬声器里出来，而右扬声器静悄悄。在现实生活中，如果声源在你的左边，你的两只耳朵都会听到，只是右耳听到的声音来得迟一点儿并且柔和一些，你的大脑根据这些信息来推定声音来自左边。3D 音效将会模仿你在现实世界中听到的声音，所以声音仍会从右扬声器中出来，只是轻柔一些，并推迟一点点(根据声音感知的数学理论)。所以，事实上是两只扬声器联合起来玩点花招欺骗你的大脑，让它以为声音来自左方。这是 3D 音效的一个重要概念，它不仅仅是调整声音的音量并将其在左扬声器和右扬声器中作分配，而且是在模拟现实世界的声波。

13.1.2 多普勒效应

如果从来没有听说过这个术语，那么请想象你正站在铁轨边，远远看见一辆火车向你驶来。它跑得很快，在它快要经过你时，以一个恒定的音阶拉响了汽笛。当火车经过你时，你会发现汽笛声的音阶听起来在下降。这就是多普勒效应，因澳大利亚数学和物理学家 Christian Doppler 而得名，他在 19 世纪发现了多普勒效应并最早为其建立了数学模型。

DirectX Audio 可以模拟多普勒效应，但除了位置之外，它还需要关于缓冲区和听者的速度方面的信息。

提示

听听音频剪辑 13.1 以感受多普勒效应。

13.1.3 缓冲区和听者的属性

现在读者已经了解了这些概念,下面是 DirectX Audio 用以正确模拟 3D 音效的主要属性变量。

(1)位置

对于初学者而言,听者和缓冲区都有位置,即一个由三元坐标(x,y,z)表示的立体空间世界中的点。注意这个位置没有刻度或中心坐标,你只是简单地告诉 DirectX Audio,“某声音缓冲区位于空间世界的(50,-50,10)”。

(2)速度

听者和缓冲区同样具有速度。它们是由带有量值(magnitude)的方向向量表示的。在实践中,可以用对象在当前帧和上一帧中的位置来构造速度向量。

(3)最小和最大距离

一个 3D 音效缓冲区有两个属性,其中一个用于指定最远可以在多远听到它,另一个用于指定当离它多近时它达到最大音量。这两个属性是最小和最大距离(仅限于缓冲区),如图 13.1 所示。

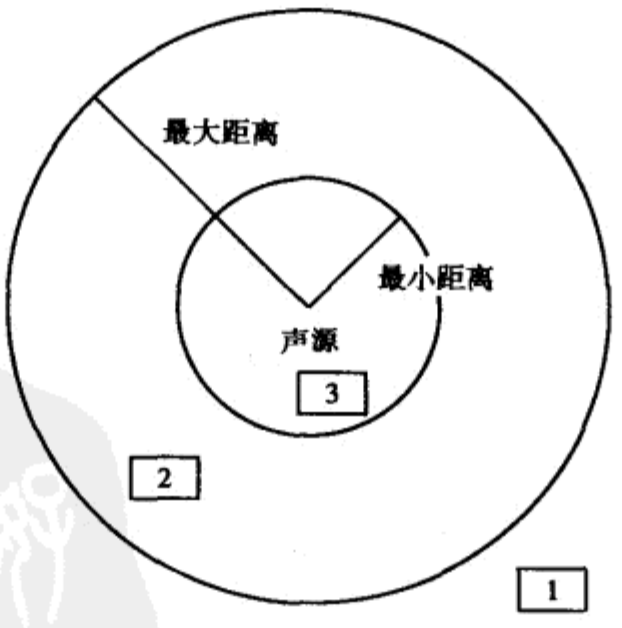


图 13.1 最小和最大距离告知 DirectX Audio 多远可以听到声音

(注:在位置 1 的听者听不见声音,在位置 2 的听者听到的是衰减后的声音,在位置 3 的听者以最大音量听到声音。)

最小距离指出当你离缓冲区多近时获得其最大音量。即使你凑得更近，DirectX 也不会将缓冲区的音量放得更大。相似地，最大距离是声音完全消失的最近距离。如果你比最近距离稍稍近一点，你能听到一点点儿声音；如果你再远一点，你将什么都听不到。如果想了解 DirectX Audio 如何在缓冲区的最小和最大音量之间插值，可以查看距离因子一节。

(4)方向(仅限于听者)

DirectX Audio 的听者有一个方向属性，这样 DirectX Audio 才会知道哪一个方向是“前方”，才可以正确地抑止(muffle)来自后面的声音。方向由两个向量指定：一个上向量(up vector)和一个前向量(front vector)。如果你的角色是一个 FPS 射手，那很简单，从瞄准器中获取方向向量即可。

(5)声音锥体(仅限于缓冲区)

缓冲区是没有方向属性的，但它们有称之为声锥(sound cones) 的传递范围。一个声锥类似于一个方向灯的灯锥。基本上需要定义一个内锥(inner cone)和外锥(outer cone)。站在内锥中的听者将会在内锥中听到基于他所处的距离的最大声音，如图 13.2 所示。

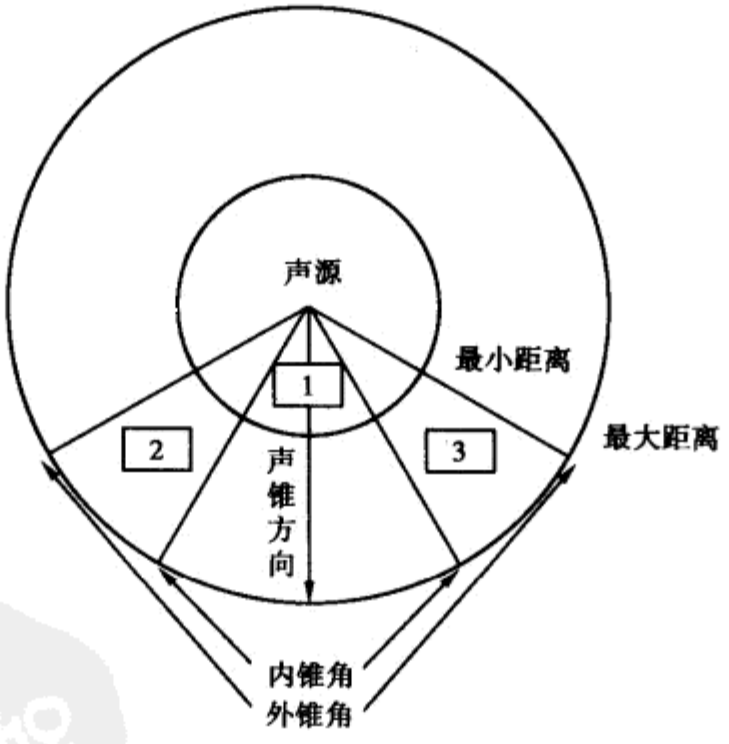


图 13.2　不同缓冲区的属性将影响声音的大小

(注：位于位置 2,3 的听者将会听到衰减后的声音，因为即使他们处于(事实上不处于)最小距离之内，他们仍处于内锥之外。只有位于位置 1 的听者才能以最大音量听到声音。)

如果听者的位置位于内锥和外锥之间，那么他所听到的声音将经过两次音量削减：一次削减取决于他离物体的距离，另一次则取决于他所处的内外锥间的位置。事实上，音量

的削减可能会更多,其原因比如听者的方向,以及一些其他因素。但基本思想是,听者位于内外锥之间的位置,是决定大致音量的一个更重要因素。

(6)流逝因子(仅限于听者)

流逝因子(rolloff factor)是听者特有的属性(缓冲区没有流逝因子属性)。它描述了所有缓冲区的衰减,换句话说,当听者离开缓冲区时声音以多快的速度减弱,或者当听者靠近缓冲区时声音以多快的速度增强。DirectX Audio 的流逝因子是一个乘数因子,它描述 DirectX Audio 与真实世界的差异程度。如果流逝因子为 1,那么游戏中的音量与现实世界中的音量相同。小于 1 的流逝因子(但大于 0,负数是非法的)表明声音在 DirectX Audio 里的衰减慢于现实世界。换句话说,0.5 的流逝因子意味着可以听到更远的声音。特别地,假设在真实世界里,一个声音在 40 米外便完全消逝,但是在 DirectX Audio 的虚拟世界里,它在 80 米之外才完全消逝。如果流逝因子为 0,则音量不随距离而改变,要么听得到,要么听不到。

(7)距离因子(仅限于听者)

对于听者而言,还有一个属性叫距离因子(distance factor)。类似于流逝因子,距离因子是游戏中关于度量单位的乘子。默认地,DirectX Audio 假设距离单位为米,即,如果听者位于(10,0,0),缓冲区位于(40,0,0),DirectXAudio 会将其解释为“缓冲区离听者 30 米远”。必须确定如果 DirectX Audio 认为相距 30 米,那么在屏幕上它们也应该相距 30 米。当然,也可以用米为单位设置游戏世界,但如果之前已经用其他现实世界中的单位设置了游戏世界,那么就必须用适当的调整因子使其一致。例如,假设所有的模型都以英尺为单位(即一个码尺的两个本地坐标为(-1.5,0,0)和(1.5,0,0)),则应设置距离因子为 0.304 8(1 英尺 =0.304 8 米)。距离因子的引入主要是为了 DirectX Audio 计算多普勒效应。

提示

初想起来,DirectX Audio 把距离和流逝因子作为听者而不是缓冲区的属性,是不可思议的。然而,为什么要以缓冲区为基础设置距离因子,使得一些缓冲区的声音比其他缓冲区传得远呢?

要模拟这种结果,应该使用缓冲区的声音锥体。要把距离因子和流逝因子看作是你在必要时可以使用的全局比例系数。可以使用距离因子以避免调整图形,也可以使用流逝因子以达到特殊目的。比如,玩家发现了"灵敏听觉头盔"并获得能力增长,则可以将流逝因子降低到接近于 0 以模拟超人听力。

(8)多普勒因子(仅限于听者)

就像距离和流逝因子一样,多普勒因子是多普勒效应的一个乘数。在不创建一个太大的世界并将快速移动的物体置于其内的前提下,多普勒因子很有用,它可以帮助你建立速度的感觉。你可以建立一个小的世界,里面的物体移动得不快,然后调整多普勒因子,使它们听起来跑得很快。

(9)处理模式(仅限于缓冲区)

DirectX Audio 允许用两种方式给缓冲区设置属性:一般模式和相对模式(head-relative mode),可以为每个缓冲区指定不同的模式。在一般模式下,指定的是空间世界中的位置和方向,这很容易。在相对模式下,距离是相对于听者而言的,这意味着,如果听者位于(7,3,0)位置,同时你告知 DirectX Audio 有一个缓冲区位于相对模式下的(-3,5,0)位置,即缓冲区位于听者的左边 3 个单位距离,上面 5 个单位距离处。DirectXAudio 经过计算可知此缓冲区位于事实世界的(4,8,0)处,如图 13.3 所示。

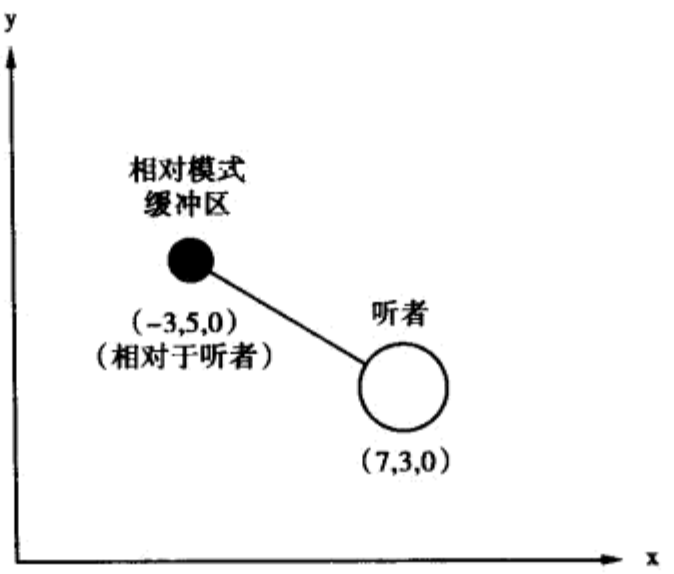


图 13.3 在相对模式下,缓冲区的位置是相对于听者而言的

(注:在相对模式下,缓冲位置是相对于听者的。在这个例子中,缓冲的最终位置是(4,8,0)。如果听者在(100,200,0)处,则最终位置将是(197,205,0)。)

选择何种模式取决于所期望的音效。对于那些始终与听者相关连的缓冲区而言,使用相对模式较为容易(想象一下这些东西直接拴在玩家的脑袋上)。比如,在一个简单的模拟飞行游戏中,可以将飞机的引擎设定为相对模式,前提是玩家永远不能旋转飞行员的头,而仅能旋转飞机本身。如果玩家在游戏中能够旋转飞行员的头,那么就不能让引擎使用相对模式。

13.1.4 立即 vs 延时设置

每一次改变缓冲区或者听者的属性，DirectX Audio 都会立即重新计算每个元素并重新混制声音输出。如果修改的是一大堆缓冲区的话，这将导致巨大的资源浪费，因为每改变一个缓冲区，DirectX Audio 就会全部重新计算！

要改变这种状况，需要使用延时设置（deferred settings）。在延时设置下，告知 DirectX Audio 在设置完所有参数之前，不要重新计算。如果修改的是所有的缓冲区和听者的话，一定要这么做。本章的示例代码会演示如何使用延时设置：调用函数设置缓冲区或听者属性时应带上 DS3D_DEFERRED 标志，然后在最后调用 CommitDeferredSettings 以应用这些变化。

13.2 3D 音效编程

理解了 3D 音效的概念之后，用 DirectX Audio 实现 3D 编程并不十分困难。下面讲解如何改进音频引擎使其支持 3D 音效。

13.2.1 C3DSoundListener

听者由一个新类 C3DSoundListener 来描述。下面是类定义：

```
class C3DSoundListener
{
public:
   friend class CAudioManager;
      virtual ~C3DSoundListener( );
   void SetPosition(D3DVECTOR position){ m_Props. vPosition = position; RefreshProps( ); }
   D3DVECTOR GetPosition( ){ return(m_Props. vPosition); }
   void SetVelocity(D3DVECTOR Velocity){ m_Props. vVelocity = Velocity; RefreshProps( );}
   D3DVECTOR GetVelocity( ){ return(m_Props. vVelocity); }
   void SetOrient(D3DVECTOR OrientTop,D3DVECTOR OrientFront){
      m_Props. vOrientTop = OrientTop;
      m_Props. vOrientFront = OrientFront;
      RefreshProps( );
   }
   D3DVECTOR GetOrientTop( ){ return(m_Props. vOrientTop); }
```

```
    D3DVECTOR GetOrientFront( ) { return( m_Props. vOrientFront) ; }
    void SetDistanceFactor( float f) { m_Props. flDistanceFactor = f; RefreshProps( ) ; }
    float GetDistanceFactor( ) { return( m_Props. flDistanceFactor) ; }
    void SetRolloffFactor( float f) { m_Props. flRolloffFactor = f; RefreshProps( ) ; }
    float GetRolloffFactor( ) { return( m_Props. flRolloffFactor) ; }
    void SetDopplerFactor( float f) { m_Props. flDopplerFactor = f; RefreshProps( ) ; }
    float GetDopplerFactor( ) { return( m_Props. flDopplerFactor) ; }
protected:
    C3DSoundListener( ) ; // 仅可被友类访问 (如,CAudioManager)

    CAudioManager * m_Manager;
    void RefreshProps( ) ;
    DS3DLISTENER m_Props;
    IDirectSound3DListener8 * m_Listener;
} ;
```

这里有很多访问器函数用以访问 m_Props 结构的成员。M_Props 结构类型为 DS3DLISTENER 类型,这种类型由 DirectX 提供并包含了听者的所有属性。

除了 m_Props 成员和相关的访问器函数之外,C3DSoundListener 类还包含一个函数 RefreshProps,一个指向 CAudioManager 的指针,以及一个指向 DirectSound 听者接口的指针 m_Listener。

注意,每个 set 访问器函数设定好新值之后都会调用 RefreshProps 函数。RefreshProps 会调用听者接口的 SetAllParameters 函数,将 m_Props 结构传递给 DirectX Audio,其代码如下:

```
void C3DSoundListener::RefreshProps( )
{
    HRESULT hr = m_Listener -> SetAllParameters( &m_Props, DS3D_DEFERRED) ;
    ThrowIfFailed( hr, "C3DSoundListener::RefreshProps( ) : Error calling SetAllParameters") ;
}
```

在这个短小的方法里惟一有趣的事是 DS3D_DEFERRED 标志的运用。这个标志告知 DirectX Audio 在听者接口的 CommitDeferredSettings 方法被调用之前,不要重新计算缓冲区。

13.2.2 CAudioManager 的修改

由于只允许有一个听者存在,作者修改了音频管理器,使得其包含并维护一个全局听

者实例。于是 CAudioManager 便有了一个名为 m_Listener 的成员变量,其类型是 C3DSoundListener。其他代码可以通过 GetListener 访问器函数访问这个听者成员。

CAudioManager 也负责在最开始时获取听者的接口。当它得到接口后,便把新的听者实例赋值给 m_Listener 成员。

以下代码属于 CAudioManager 的 Init 函数的一部分:

```
// 获取听者接口
IDirectMusicAudioPath8 * defaudiopath = NULL;
IDirectSound3DListener8 * listener;
hr = GetPerformance( ) -> GetDefaultAudioPath( &defaudiopath) ;
ThrowIfFailed( hr," CAudioManager::Init: can't get default audio path!" ) ;
hr = defaudiopath -> GetObjectInPath( 0, DMUS_PATH_PRIMARY_BUFFER, 0,
   GUID_NULL, 0, IID_IDirectSound3DListener8, ( LPVOID * ) &listener) ;
ThrowIfFailed( hr," CAudioManager::Init: can't get listener on default audio path!" ) ;

// 告知听者对象使用此听者接口
GetListener( ). m_Listener = listener;
GetListener( ). RefreshProps( ) ;
Commit3DSoundParameters( ) ;
SAFE_RELEASE( defaudiopath) ;
```

这段代码先获取默认音频路径的接口,再调用 GetObjectInPath 方法获取听者的接口。由于在 DirectX Audio 里,只有一个听者,所以无论从哪个音频路径去获取听者的接口,结果都一样。

取得听者接口后,代码就将其赋值给 C3DSoundListener 对象,并调用其 RefreshProps 方法,这样对象便有机会将其默认参数传递给 DirectX Audio。同时,由于听者的 RefreshProps 方法使用延时标志设置参数,代码立即调用 Commit3DsoundParameters 函数以强制 DirectX Audio 接受所有新设定的参数并重新计算输出。这些工作都完成后,代码便释放音频路径的接口。需要注意的是,代码并没有释放听者的接口;此时,听者对象占有着听者接口,直到听者的析构函数被调用时才会将其释放。

Commit3DsoundParameters 函数只有一行代码:

```
void CAudioManager::Commit3DSoundParameters( ) {
    GetListener( ). m_Listener -> CommitDeferredSettings( ) ;
}
```

这个函数仅调用听者接口的 CommitDeferredSettings 函数以提交参数。

13.2.3 C3DSoundEffect

C3DSoundEffect 是另外一个新类,它表示一个 3D 音效缓冲区:

```
class C3DSoundEffect : public CDirectMusicSegment
{
public:
    C3DSoundEffect(CAudioManager * mgr);
    virtual ~C3DSoundEffect();
    // 覆盖段的函数以播放我们自己的音频路径
    virtual bool Play(){ return(CDirectMusicSegment::Play(NULL, false, m_AudioPath)); }
    virtual bool Play(CSoundInstance * newinst,bool control)
      {return(CDirectMusicSegment::Play(newinst,control,m_AudioPath)); }
    virtual bool PlayAsPrimary(CSoundInstance * newinst){
      return(CDirectMusicSegment::Play(newinst,true,m_AudioPath));
    }
    // 属性
    // 略去访问器函数以节省篇幅
  protected:
    void RefreshProps();
    DS3DBUFFER m_Props;
    IDirectMusicAudioPath8 * m_AudioPath;
    IDirectSound3DBuffer8 * m_3DBuffer;
};
```

同样,这个类的大部分操作都是访问 m_Props 结构(这个类是 DS3DBUFFER 类型)的访问器函数。除此之外,还有一些新东西。

首先,注意到 C3DSoundEffect 继承于 CDirectMusicSegment 类,但它重写了 Play 函数以屏蔽 CDirectMusigSegment 的 Play 函数。之所以这么做,是因为 3D 音效缓冲区不同于一个普通的段,它每次播放声音时都需要明确地提供一个 3D 音频路径。

这意味着需要回到 CDirectMusicSegment 类并修改它,使得你可以传入自己的音频路径指针。但如果这样做的话,最终创建的 CSoundInstance 将不会释放音频路径指针。这里不再逐行说明,因为它们都是不言自明的,但是读者确实应该知道这一切是怎么发生的。

C3DSoundEffect 类需要自行创建和管理音频路径,下面是音频路径的创建过程(摘抄自 C3DSoundEffect 的构造函数):

```
// 创建自己的 3D 音频路径
HRESULT hr = m_Manager -> GetPerformance( ) -> CreateStandardAudioPath(
  DMUS_APATH_DYNAMIC_3D, CAudioManager::NUMCHANNELS,
    TRUE, &m_AudioPath);
ThrowIfFailed(hr, " C3DSoundEffect: couldn't create 3D audio path!");

// 从新的音频路径获取 3D 音频缓冲区
hr = m_AudioPath -> GetObjectInPath( DMUS_PCHANNEL_ALL,
  DMUS_PATH_BUFFER, 0, GUID_NULL, 0,
  IID_IDirectSound3DBuffer8, (LPVOID *) &m_3DBuffer);
ThrowIfFailed ( hr, " C3DSoundEffect: couldn't get 3D DirectSound buffer from audio
path!");
```

这段代码创建了一个标准的 3D 音频路径(注意 DMUS_APATH_DYNAMIC_3D 标志的使用),然后再向其查询内置的 3D DirectSound 缓冲区的接口。

C3DSoundEffect 也有一个与听者相类似的 RefreshProps 函数,区别仅在于它操作的是一个 DirectSound 缓冲区,而不是一个听者。

13.3 外层空间的示例程序

本小节中,作者希望创建一个可以演示真实的 3D 世界里声音效果的示例程序。当然,本来可以仅提供一个控制台应用,但它不足以展示如何将 3D 音效与游戏相集成。要完全理解 3D 音效,需要察看原代码,它与 3D 图形代码一起工作以创建一个令人信服的现实世界。

Ch13p1 示例是在作者写的前一本书《DirectX 8.0 游戏特效编程》中的一个示例基础上修改而来的,它是那本书第 20 章 Ch20p3_ComplexExplosion 示例的简单修改版本。

本书不准备花时间去解释示例中的 3D 图形部分是怎么工作的,但要正确理解示例程序,需要知道诸如太空盒(skyboxes)、工作转换矩阵(work transform matrics)、及 X 文件陷阱(X file meshes)等概念。

运行这个应用,将会看到如图 13.4 的画面。

假设你身处外层空间,并且可以自由地往各个方向移动。在你的前方是一艘太空梭,它发出两个 3D 声音:不间断的发动机声音(engine. wav)及周期性发出的嘟嘟声(beep. wav)。

敲击 W、A、S 及 D 键可以让你向前、向左、向后、向右移动。而 Q 键及 Z 键可以让你上



图 13.4　本章示例程序的一个截图

下移动，B 键可以触发太空梭的嘟嘟声，G 键可以触发快速驾驭（允许你行进更快，更容易感受到多普勒效应）。

当你在示例程序中飞来飞去时，可以一直听到 3D 声音。对于初学者，可以留意，当你离飞船更近时，音量会增加，它的声音排位使人觉得声音确实来自太空梭。还要注意到多普勒效应，在你飞近、经过、并离开飞船时，可以听到发动机及嘟嘟声的音调发生变化。

Ch13p1_3DSound 示例程序是基于 D3D 示例程序框架写就的，即程序中的所有动作都发生在 CMyD3DApplication 对象里。D3D 示例应用程序框架会在合适时机调用这个对象的不同成员。

13.3.1　装载声音

在 CMyD3DApplication 对象的 OneTimeSceneInit 函数里，代码会从硬盘加载 WAV 文件，并设置它们的默认行为：

```
m_Beep = GetAudioManager( ) -> Load3DSound( "beep. wav" );
m_Engine = GetAudioManager( ) -> Load3DSound( "engine. wav" );
```

```
CDirectMusicSegment * engineseg =
    dynamic_cast < CDirectMusicSegment * > (m_Engine. Get( ) );
engineseg -> SetRepeats( DMUS_SEG_REPEAT_INFINITE) ;

// 设置引擎和嘟嘟声的范围
C3DSoundEffect * beep = dynamic_cast < C3DSoundEffect * > (m_Beep. Get( ) );
C3DSoundEffect * engine = dynamic_cast < C3DSoundEffect * > (m_Engine. Get( ) );
if (beep) {
    beep -> SetMaxDistance(100) ;
    beep -> SetMinDistance(10) ;
}
if (engine) {
    engine -> SetMaxDistance(80) ;
    engine -> SetMinDistance(10) ;
}
GetAudioManager( ) -> Commit3DSoundParameters( ) ;
```

这段代码将两个 WAV 文件装载到 3D 音效 m_Engine 及 m_Beep。然后,代码给两个对象设置一些 3D 音效参数。注意,根据这些参数,当距离太空梭的距离超过 80 个单位时,引擎声消失,而嘟嘟声却能再传 20 个距离单位。当你距太空梭 10 个单位距离内时,两个音效都达到最大音量。

同时也要注意代码将 m_Engine 强行转换为 CDirectMusicSegment 指针,然后再调用一个名为 SetRepeats 的新操作,将其设置为无限循环。SetRepeats 仅仅是 IDirectMusicSegment 同名函数的包装函数。

参数设置完成后,代码将指示音频管理器提交这些参数,一切就绪,游戏循环就可以开始了。

13.3.2 更新缓冲区和听者

每一帧,缓冲区和听者的位置及相关属性都需要被更新。以下是更新代码,它摘抄自 CMyD3DApplication 类的 FrameMove 函数:

```
m_BeepTimer += m_fElapsedTime;
if( m_BeepActive && m_BeepTimer > BEEPINTERVAL) {
    // 发出嘟嘟声!
    m_Beep -> Play( ) ;
```

```
    m_BeepTimer = 0;
  }
  // 设嘟嘟声的位置为太空梭的位置
  C3DSoundEffect * beep = dynamic_cast < C3DSoundEffect * > (m_Beep.Get());
  if (beep) {
    D3DVECTOR v;
    v.x = 0;
    v.y = 4;
    v.z = 0;
    beep -> SetPosition(v);
  }
  // 设听者的位置为摄像机的位置
  C3DSoundListener &listener = GetAudioManager() -> GetListener();
  listener.SetPosition(m_Camera.GetPosition());
  listener.SetDopplerFactor(DS3D_MAXDOPPLERFACTOR);
  listener.SetDistanceFactor(10);
  // 设置听者方向
  D3DXVECTOR3 lookat = D3DXVECTOR3(
      m_Camera.GetViewMatrix()._13,
      m_Camera.GetViewMatrix()._23,
      m_Camera.GetViewMatrix()._33);
  D3DXVECTOR3 up = D3DXVECTOR3(
      m_Camera.GetViewMatrix()._12,
      m_Camera.GetViewMatrix()._22,
      m_Camera.GetViewMatrix()._32);
  D3DXVECTOR3 nlookat,nup;
  D3DXVec3Normalize(&nlookat,&lookat);
  D3DXVec3Normalize(&nup,&up);
  if (nup == nlookat) { nup = D3DXVECTOR3(0,1,0); }
  listener.SetOrient(nup,nlookat);
  // 根据上一帧和本帧的位置差异设置听者的速度
  m_ListenerVelocity.x = m_Camera.GetPosition().x - m_LastListenerPosition.x;
  m_ListenerVelocity.y = m_Camera.GetPosition().y - m_LastListenerPosition.y;
```

```
m_ListenerVelocity.z = m_Camera.GetPosition().z - m_LastListenerPosition.z;
listener.SetVelocity(m_ListenerVelocity);
m_LastListenerPosition = m_Camera.GetPosition();
GetAudioManager() -> Commit3DSoundParameters();
```

这段代码完成了很多不太复杂的任务。首先,它先递增一个计时器的值并判定是否需要播放嘟嘟声。然后,它将嘟嘟声的3D音效缓冲区位置移到太空梭所在的位置(硬编码为(0,4,0))。然后,代码为听者做同样的工作,并接下来设置听者的方向属性。

幸运的是,摄像头(camera)类包含一个视野矩阵(view matrix),只需要获取上方(up)及观察方(look at)向量,将它们标准化,再调用SetOrient设置听者的上方(up)及观察方(look at)向量即可,这个过程很容易。接着,代码比较当前帧与上一帧的相机位置以计算听者的速度。

最后,还有一些人造效果。代码将多普勒因子设到最大,设距离因子为10,以让多普勒效应显现出来(这样你就可以听到它们)。你应该试验这两个参数的不同值组合,以感受应当如何创建游戏的特定听觉环境。代码最后提交了所有参数的改变值。

13.4 本章小结

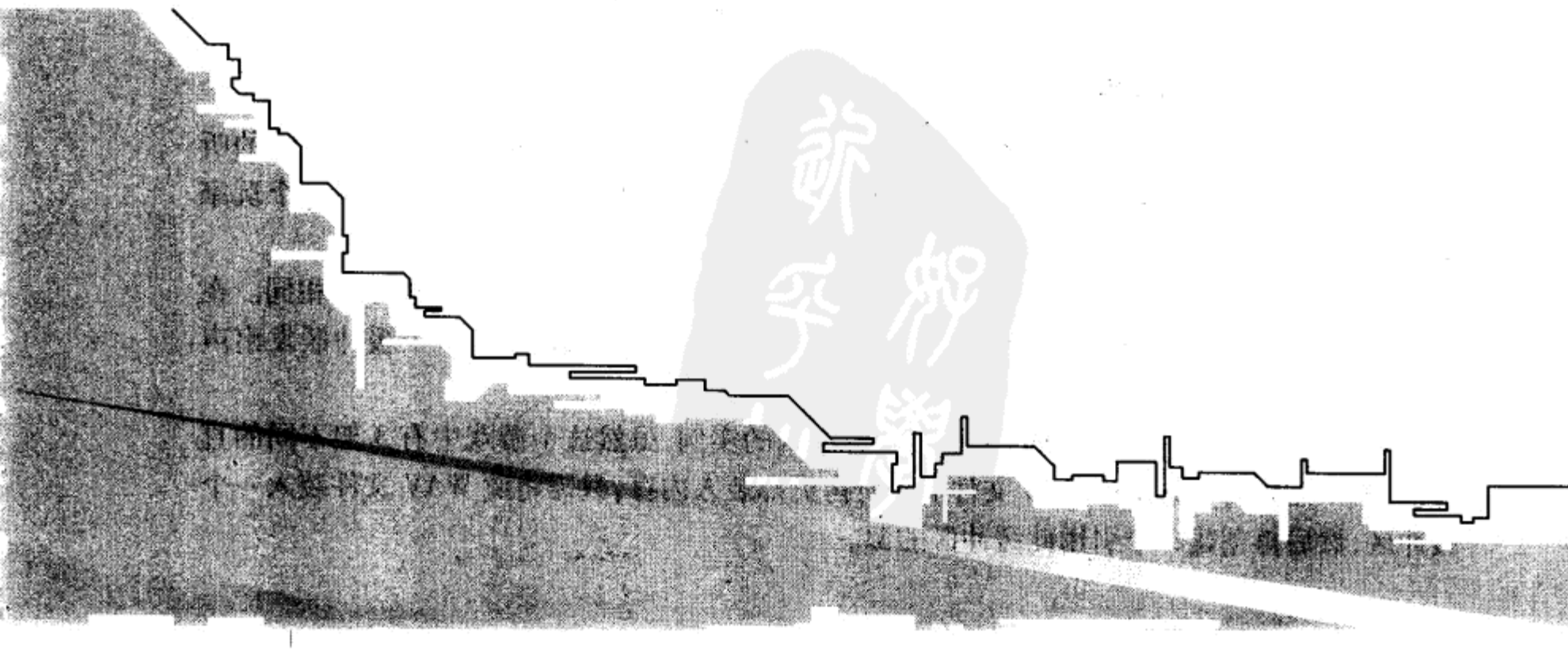
实施3D音效最复杂的事情在于学习3D音效背后的概念,以及如何通过DirectMusic及其接口直接操作3D DirectSound缓冲区。通过学习这些概念和技术,在实现真实的3D世界中的音效时,就不再需要编写成堆的代码以执行复杂的数学计算,但这并不是说数学过程没有价值。如果有兴趣,可以查看光盘中提供的链接,以获取更多的信息。

现在已经对3D音效有了一个基本的了解。在下一章,我们将继续扩充相关知识,读者们将学到如何应用第三方API进行高级3D音效的处理,如障碍和反射效果。

PDG

第 14 章

OpenAL 中的 3D 音效



通过上一章的学习，相信读者已经获得了用 3D 音效创建真实世界的体验。虽然那是仅有一个对象的贫乏世界，但读者已知道了它的基本思想。

本章将继续同样的工作，区别在于这里采用的是 OpenAL 库、或者开放音频库（Open Audio Library），而不是 DirectX。

OpenAL 是一个由个人和企业组成的协会创立的用于创建 3D 音效的开放式、跨平台的 API。它是支持 3D 图形渲染的 OpenGL 的姊妹版本，它的设计与体例与 OpenGL 基本相同。

CreativeLabs 公司完成了 OpenAL 的大部分工作，他们与一家现在已经破产的 Loki Entertainment Software 的公司合作，为 OpenAL 的推广做出了积极的努力。今天，我们可以从他们的网站（developer. creative. com）上找到 OpenAL 的 SDK。

OpenAL 是一个设计良好的跨平台的 API，它被专门设计用于支持 3D 音效。如果不准备采用动态音乐，或者不介意自行书写代码让 OpenAL 支持动态音乐，那么就可以用 OpenAL 完成所有的游戏音效。

14.1 OpenAL 与 DirectX Audio 中 3D 音效的区别

如果对第 11 章的概念还没有理解，请先返回去学习那些概念。OpenAL 中 90% 的概念与 DirectX Audio 3D 音效的概念是相同的。下面逐一加以介绍。

14.1.1 源、缓冲区和听者

OpenAL 的功能涉及到 3 种类型的对象：源（sources）、缓冲区和听者。OpenAL 中的听者在概念上与 DirectXAudio 中的相同：在 OpenAL 中，仍然只有一个听者，它代表一个玩家的耳朵。

然而，在 OpenAL 中增加了一个称为“源”的新对象，而且缓冲区的概念也不尽相同。在 OpenAL 体系中，缓冲区是一个 PCM 样本，而源描述一个缓冲区的实例，一个缓冲区发出声音的位置。

图 14.1 给出了一个利用 OpenAL 的这种结构的实例，虽然这个游戏中有 3 架不同的直升机，但不需要将 helicopter. wav 文件的 3 份拷贝都装入内存，只需将此 WAV 文件装入一个缓冲区，然后再创建 3 个引用此缓冲区的源。

从不同角度看，OpenAL 和 DirectX Audio 中的 API 都具备这种能力。在 OpenAL 里，多个源可以引用同一个缓冲区，而缓冲区包含了事实上的语音数据。在 DirectX Audio 里，这项工作由音频路径完成。可以这样理解，OpenAL 的缓冲区等价于 DirectX 的段，而 OpenAL 的源相当于 DirectX 的音频路径。

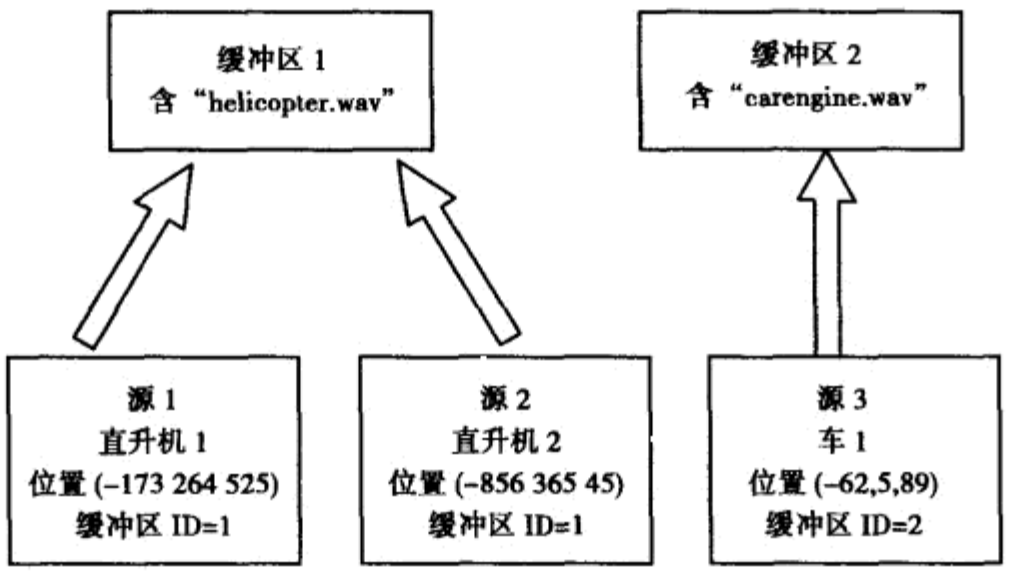


图 14.1 多个源可以引用同一个缓冲区

14.1.2 不再是 COM

DirectMusic 的 API 是由 COM 驱动的，也就是说做所有工作都要获取和释放接口。但 OpenAL 已不再由 COM 驱动，它是跨平台的，且基于一个经过检验证明可靠的 API 接口，它是包含多个函数、枚举、结构的表。

OpenAL(OpenGL 也一样)的函数命名遵循严格的命名规范。如果查看 API 文档，你将发现有些函数的名称几乎完全一样，它们只有结尾不同，一个函数以 f 结尾，另一个以 fv 结尾，第三个则可能以 i 结尾。这些不同的结尾指明函数参数的不同数据类型。

例如，有 4 个分别名为 alListenerf、alListener3f、alListenerfv 及 alListeneri 的函数(顺便强调一下，OpenAL 中所有函数都以 al 打头)。这 4 个函数完成相同的工作——设置听者的属性。他们所不同的是，alListenerf 接受 1 个浮点参数，alListener3f 接受 3 个浮点参数(1 个向量)，alListenerfv 接受 1 个浮点数组(C ++ 术语里也叫向量)，alListeneri 接受 1 个整型参数。这看起来很奇怪，但只要懂得他们的含义后便会习惯，它确保数据类型不会弄错。

14.1.3 缓冲区属性

所有缓冲区的可用属性都列于表 14.1 中。正如所看到的一样，它们都与装入缓冲区的 WAV 文件的属性相关。

表 14.1 OpenAL 的缓冲区属性

属性	描述
AL_FREQUENCY	WAV 文件的频率
AL_BITS	WAV 文件的每个样本比特数
AL_CHANNELS	声道的数量(1 = 单声道,2 = 立体声)。注意,如果声道的数目大于1,则此缓冲区被播放时不会被定位(换句话说,如果希望声音来自正确的位置,请使用单声道)
AL_SIZE	缓冲区的大小(以字节计)
AL_DATA	数据最初装入时的来源内存地址(不是特别有用,因为在将数据装入缓冲区后那块内存多半已经释放)

所有这些属性都是只读的,它们的值在调用 alBufferData 将样本数据装入缓冲区时被设定。这些属性值都是整数,可以用 OpenAL 函数 alGetBufferi 去获取这些值。

14.1.4 源的属性

OpenAL 中的源有很多属性,见表 14.2。其中大部分已在上一章中见过,但也有几个新属性。

表 14.2 OpenAL 的源属性

属性	描述
AL_PITCH	这个源的一个音调乘数(总大于0)。介于0与1间的乘数将导致音调降低,大于1的乘数则提高音调
AL_GAIN	源的音量乘数。介于0~1的乘数降低音量,大于1的乘数提高音量
AL_MAX_DISTANCE	最大距离,大于此距离听者将无法听到源的声音
AL_ROLLOFF_FACTOR	源的流逝因子。通常为1,也可以降低到0~1之间以获得更远的流逝距离,或者提高到1之上以减短流逝距离

（续表）

属性	描述
AL_REFERENCE_DISTANCE	源失去一半音量时的距离（不考虑流逝因子）
AL_MIN_GAIN	源的最小音量
AL_MAX_GAIN	源的最大音量
AL_CONE_OUTER_GAIN	听者位于源的声锥之外时的音量
AL_CONE_INNER_GAIN	听者位于源的声锥之内的音量
AL_CONE_OUTER_ANGLE	声锥的角度。默认值是 360，即没有声锥
AL_POSITION	由 3 个浮点数组成的向量，表示源的位置
AL_VELOCITY	由 3 个浮点数组成的向量，表示源的速度（用于多普勒效应）
AL_DIRECTION	由 3 个浮点数组成的向量，表示源的声锥所指的方向
AL_SOURCE_RELATIVE	如果此属性设为 AL_TRUE，则源的位置是相对于听者而言的（类似于 DirectX Audio 中的（听者）相对模式）
AL_LOOPING	如果此属性设为 AL_TRUE，则源会循环播放
AL_BUFFER	源播放的缓冲区的 ID 号
AL_SOURCE_STATE	源的当前状态（AL_INITIAL，AL_STOPPED，AL_PLAYING，AL_PAUSED）
AL_BUFFERS_QUEUED	在这个源上排队等待播放的缓冲区的个数（这个属性是只读的。只可以通过调用 alSourceQueueBuffers 间接设定它）
AL_BUFFERS_PROCESSED	在源上所有排队的缓冲区中已被播放的个数（同样，这也是一个只读属性）

可以通过 OpenAL 函数 alSourcef、alSourcefv、alSource3f 及 alSourcei 设定源的属性，也可以根据数据类型的不同，用 alGetSourcef、alGetSourcefv、alGetSource3f 及 alGetSourcei 获取属性的值。

提示

要小心——OpenAL 使用左手习惯的坐标系统，而 Direct3D 的坐标系统是右手习惯的。换种方式说，在 OpenAL 里，负的 z 值表示向屏幕下方移动，而在 Direct3D 中，负的 z 值在屏幕之外。

14.1.5 听者的属性

OpenAL 的听者也有一系列属性，见表 14.3。可以调用 alListener 族的函数（包括 alListenerf、alListenerfv、alListener3f 及 alListeneri）设定听者的属性。获取听者属性可以调用 alGetListenerf、alGetListenerfv、alGetListener3f 及 alGetListeneri 函数。

表 14.3 OpenAL 的听者属性

属性	描述
AL_GAIN	主音量（这相当于音量旋钮；把其值设为大于 1，则所有的声音都会被放大，设其值在 0～1 之间，则所有的声音都会降低音量）
AL_POSITION	由 3 个浮点数组成的向量，表明听者在 3D 世界中的绝对位置
AL_VELOCITY	由 3 个浮点数组成的向量，表明听者的速度，用于多普勒效应
AL_ORIENTATION	两个向量，由 6 个浮点数组成的属性，以表明听者的方向。其中，前 3 个浮点数构成观察方（look at）向量，后 3 个构成上方（up）向量

14.2 将 OpenAL 集成到音频引擎

在音频引擎中加入 OpenAL 的支持十分简单明了。本节将带领读者安装 OpenAL SDK，并创建几个新的音频引擎类。

14.2.1 安装 OpenAL

要安装 OpenAL SDK，先浏览光盘中的目录 goodies，然后，双击 OpenAL goodies 目录中的 OpenALSDK.exe 文件。然后跟随安装向导，将 SDK 安装到 C:\Program Files\Creative Labs\OpenAL 1.0 SDK。

提示

如果你使用了 OpenAL,那么在你游戏的分发版本中请包含 OpenAL 的可分发文件,这些文件位于 OpenAL SDK 的 dll\Redist 文件夹中。这个分发文件包允许任何需要用到 OpenAL 的人将 OpenAL 运行时文件安装于他的系统中。

这个 SDK 包含几个示例程序和一些帮助文档,建议以后再去细看它们。现在,要使用 OpenAL,只需要确定 OpenAL 的 Include 和 Lib 目录位于项目的合适搜索路径中。本章的示例程序将会试图在..\..\OpenAL 1.0 SDK\Include 路径中寻找 Include 文件,在..\..\OpenAL 1.0 SDK\lib 中寻找 lib 文件。

14.2.2 OpenAL 库的结构

OpenAL SDK 分为两个库。其中一个是核心库(OpenAL32.lib),另一个是实用程序库(ALut.lib)。OpenAL 实用程序库是可选的,但大多数情况下都会用到它,因为它使得应用 OpenAL 相对容易一些。

OpenAL 的库函数分为 3 个类别。大多数的函数属于 OpenAL 核心类,所有 OpenAL 核心类的函数都以 al 开头。也有一些函数属于 OpenAL 实用程序类(以 alut 开头)。另外一些函数(我们在这里不讨论),它们属于 OpenAL 上下文(context)类(以 alc 开头)。当使用超过一个音频设备或者希望明确地控制惟一的音频设备时,上下文函数会很有用。大多数的游戏都不会考虑这个问题,它们依赖于 OpenAL 工具库去自动选择最好的上下文函数并使用它。

14.2.3 OpenAL 音频引擎的新类

要在音频引擎中支持 OpenAL 必须创建一系列的新类。第一个也是最重要的一个新类是 COpenALManager,其作用相当于 CAudioManager。之所以这样做是希望 OpenAL 部分使用的引擎与其他部分的 DirectX 引擎完全独立。

以下是 COpenALManager 对象类的定义:

```
class COpenALManager
{
public:
    COpenALManager();
    virtual ~COpenALManager();
    virtual void Init();
```

```
    virtual void UnInit();
    COpenALBufferPtr CreateBufferAndLoadWave(std::string filename);
    COpenALSourcePtr CreateSource(COpenALBufferPtr linktobuffer);
    COpenALListener &GetListener() { return(m_Listener); }
protected:
    static COpenALListener m_Listener;
};
```

这段代码包含了一些初始化、反初始化 OpenAL 的函数，一些创建缓冲区、源的函数，以及一个访问 OpenAL 听者的 GetListener 函数。

使用音频引擎中 OpenAL 部分的代码要负责 COpenALManager 类的实例化，并向其提供一个访问器函数（就像为 CAudioManager 提供的一样）。

其他的新类包括 COpenALBuffer、COpenALListener、COpenALSource，这些类都会在后续的章节中学到。

同时也要注意到 COpenALBuffer 及 COpenALSource 都有智能指针类：COpenALBufferPtr 及 COpenALSourcePtr。这两个智能指针与你以前见过的没什么区别，它们继承于 CRefCountPtr 模板并特指为 COpenALBuffer 及 COpenALSource 工作。没有这些智能指针前，我们需要记住在不再需要时将缓冲区及源删除时，智能指针将会自动替我们完成这些工作。

14.2.4 初始化和反初始化 OpenAL

现在已经安装了 OpenAL，并且对相关新类有了初步认识，可以书写使用它的代码了。看一看图 14.2，它描述了使用 OpenAL 的简要过程。

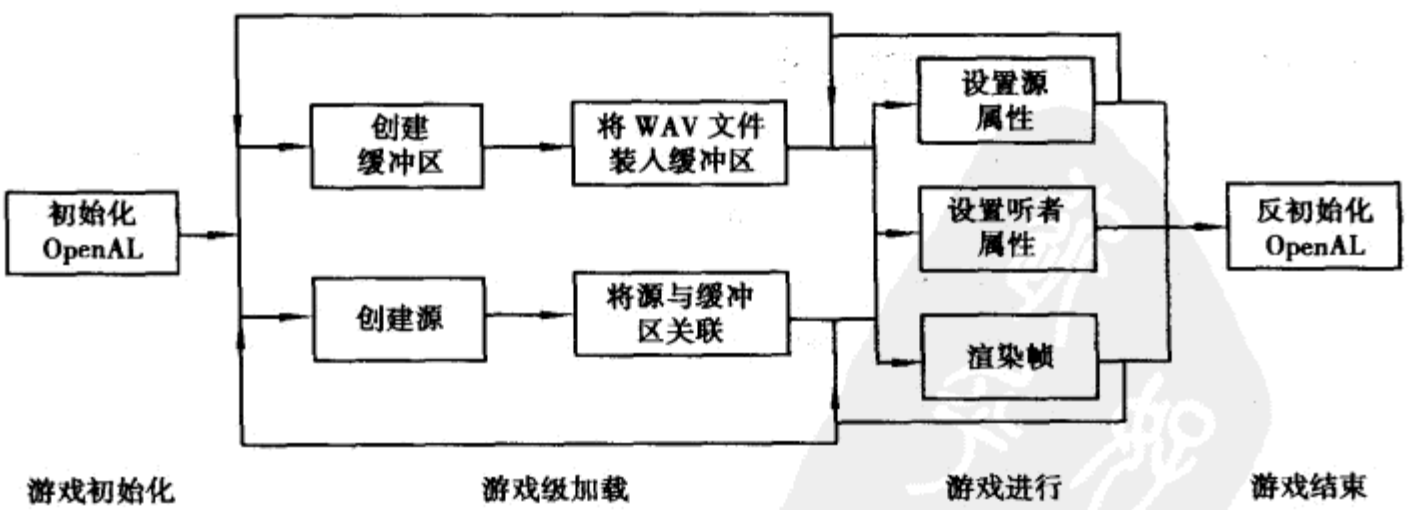


图 14.2 使用 OpenAL 的标准流程

所有使用 OpenAL 的代码都必须以初始化 OpenAL 库为开始，以反初始化 OpenAL 库为

结束。做这些事情的最简单方法是调用 OpenAL 实用程序库中的函数 alutInit 及 alutExit。下面是摘抄自 COpenALManager 的 Init 及 UnInit 的示例代码：

```
void COpenALManager::Init()
{
  alutInit(0, NULL);
  alGetError(); // 清除错误代码
}
void COpenALManager::UnInit()
{
  alutExit();
}
```

alutInit 函数有两个参数(argc 和 argv)，但是这里它们没有被用到，所以指定 argc 为 0，argv 为 NULL。alutExit 函数没有参数。

注意，代码在完成 OpenAL 初始化后立即清除错误代码值。这是我们接下来要讨论的问题，即 OpenAL 中的错误检查。

14.2.5　OpenAL 中的错误检查

调用 OpenAL 的函数来检查错误有两种主要方法。其一，可以通过一些函数的返回值来表示错误信息的值(如果函数返回 AL_FALSE，意味着有错误发生)。

其二，可以调用 alGetError 函数去判断前一件事情是否发生了错误。alGetError 返回一个枚举型值，一切正常的话这个值会是 AL_NO_ERROR，如果有错误发生，这个值就会是表 14.4 中的一项。

表 14.4　OpenAL 的错误代码

返回值	含义
AL_INVALID_NAME	输入了错误的名称
AL_INVALID_ENUM	向函数输入了一个错误的整数，函数期望一个合法的枚举型整数
AL_INVALID_VALUE	输入的值没有任何意义或者非法(比如，指定了一个负值给源的音调属性)
AL_INVALID_OPERATION	在错误的时间或者用错误的方式调用了一些方法
AL_OUT_OF_MEMORY	内存分配失败

14.2.6 创建和撤消缓冲区

要设置游戏音效,需要做的第一件事情通常是创建缓冲区并装载 WAV 文件。这项工作是由名字冗长,但用途明确的 COpenALManager 函数的 CreateBufferAndLoadWave 完成的。以下是它的代码:

```
COpenALBufferPtr COpenALManager::CreateBufferAndLoadWave(std::string filename)
{
  COpenALBuffer *newbuffer = new COpenALBuffer();
  unsigned int newid;
  alGenBuffers(1, &newid);
  if (alGetError() != AL_NO_ERROR) {
      Throw("COpenALManager::CreateBuferAndLoadWave: error generating AL buffer.");
  }
  newbuffer -> m_ID = newid;
  newbuffer -> LoadWave(filename);
  return(COpenALBufferPtr(newbuffer));
}
```

这里可以看到代码首先创建一个新的 COpenALBuffer 实例,然后再调用 alGenBuffers。alGenBuffers 函数有两个参数:一个参数指明想创建多少个缓冲区(本例为一个),以及一个指向整数数组的指针(或者仅仅是一个整数指针,如果仅创建一个缓冲区的话),这个整数数组将会接收新创建的缓冲区的 ID 值。这段代码仅创建一个缓冲区并将其 ID 值保存到 newid 变量中,这个 ID 值接下来将被传递给新的 COpenALBuffer 类。当类获得其 ID 值后,OpenAL 管理器会调用缓冲区的 LoadWave 函数,这个函数会将指定的 WAV 文件装入缓冲区中。这些工作完成后,代码为缓冲区创建并返回一个智能指针。

以下是 LoadWave 的代码:

```
void COpenALBuffer::LoadWave(std::string filename)
{
  unsigned char *data = NULL;
  alutLoadWAVFile(const_cast<char *>(filename.c_str()), &m_Format,
    (void **)&data, &m_Size, &m_Freq, &m_Loop);
  // 将 WAV 数据拷贝到缓冲区
  alBufferData(m_ID, m_Format, data, m_Size, m_Freq);
  if (alGetError() != AL_NO_ERROR) {
```

```
        alutUnloadWAV(m_Format, data, m_Size, m_Freq);
        Throw("COpenALBuffer::LoadWave: error loading WAV data into buffer.");
    }
    // 释放原始数据
    alutUnloadWAV(m_Format, data, m_Size, m_Freq);
}
```

真正的工作是由 OpenAL 实用程序库的函数 alutLoadWAVFile 完成的。这个函数接受两个输入:一个是文件名和一个是指针的地址。它分配内存,将 WAV 装入内存,并设置指针,然后返回 WAV 文件的属性。这段代码将所有的东西传递给 alBufferData 函数,此函数会将指定 ID 的 wave 数据拷入缓冲区。这些工作完成以后,代码调用 OpenAL 工具箱函数 alutUnloadWAV 删除原始 wave 数据。

当不再需要缓冲区时,与其相关联的智能指针 COpenALBufferPtr 将其自动删除。作为删除过程的一部分,COpenALBuffer 的析构函数会被调用。

析构函数的代码如下:

```
COpenALBuffer::~COpenALBuffer()
{
    if (alIsBuffer(m_ID) == AL_TRUE) alDeleteBuffers(1, &m_ID);
}
```

简单地讲,它询问 OpenAL 指定的 ID 是否还存在,如存在,调用 alDeleteBuffers 函数删除对应的缓冲区。

14.2.7 创建和撤消源

在缓冲区被创建并装好了 WAV 数据之后,需要创建一些源并将其与缓冲区关联。这些工作是由 COpenALManager 的 CreateSource 函数实现的:

```
COpenALSourcePtr COpenALManager::CreateSource(COpenALBufferPtr linktobuffer)
{
    COpenALSource *newsource = new COpenALSource();
    unsigned int newid;
    alGenSources(1, &newid);
    if (alGetError() != AL_NO_ERROR) {
        Throw("COpenALManager::CreateSource: error generating AL source.");
    }
    newsource->m_ID = newid;
```

```
    newsource -> LinkToBuffer( linktobuffer) ;
    return( COpenALSourcePtr( newsource) ) ;
}
```

这里的模式同创建一个缓冲区是一样的。这段代码创建一个新的 COpenALSource 实例,然后调用 alGenSources 以创建 OpenAL 源,获取其 ID 并传递给新类,接着调用其 LinkToBuffer 方法告知源它应该引用的缓冲区。它也是以返回一个指向新源的智能指针作为结束的。

COpenALSource 的 LinkToBuffer 方法仅仅设定源的 AL_BUFFER 属性:

```
void COpenALSource::LinkToBuffer( COpenALBufferPtr linkto)
{
    alSourcei( m_ID, AL_BUFFER, linkto -> GetID( ) ) ;
    if ( alGetError( ) != AL_NO_ERROR) {
        Throw( "COpenALSource::LinkToBuffer: error linking to OpenAL buffer. " ) ;
    }
    m_Buffer = linkto;
}
```

请注意 COpenALSource 类保持着一个指向其引用缓冲区的指针 m_Buffer。这保证在引用缓冲区的所有源没有被删除(或者引用到其他缓冲区)之前缓冲区不会被删除。再一次,智能指针使这件事情变得很容易。

源的撤消与缓冲区的撤消相似,也是由析构函数完成,析构函数调用 alIsSource 去确定源的 ID 依然有效,然后调用 alDeleteSources 将源从 OpenAL 移除。

14.2.8 播放源

调用 COpenALSource 的 PlayFromStart 函数可以播放源。同时可用其他函数暂停、继续、停止、重播源。这些函数是 alSourcePlay、alSourcePause、alSourceStop 及 alSourceRewind。这些函数的代码都很简单。

14.3 用 OpenAL 的外层空间示例程序

本章的示例程序的功能与上章相同,区别在于本章使用 OpenAL 而不是 DirectX Audio。从概念上,本游戏需要完成的工作与应用 DirectX Audio 3D 音效的示例游戏很相似。游戏用 OneTimeSceneInit 函数创建缓冲区和源,然后用 FrameMove 函数设置所有东西的位置和速度。虽然最终的函数调用有细小区别,但对照阅读本章和上章的代码,可以明白其工作

原理。可以在光盘里的 Ch14p1_3DOpenALSound 目录中找到本章的示例程序源代码，上一章的代码在 Ch13p1_3DSound 目录里。

14.4 本章小结

OpenAL 和 DirectX Audio 比较，哪一个更好呢？从细节的特征分析，两种 API 不相上下。谈到 3D 音效部分，如果 OpenAL 有一个特性，则 DirectX Audio 必然有一个相似的特征。很难从其中一个系统的 API 中找出另一个系统的 API 不具备的重要功能。

OpenAL 相对于 DirectMusic 有一项重要优势：DirectX Audio 中的 3D 音效绑定于 Windows 平台之上，而 OpenAL 现在支持 Windows 及 Macintosh，而且将来它会支持 Linux、Unix 及其他平台。如果希望你的游戏是跨平台的，并且需要 3D 音效，则 OpenAL 是更好的选择。

相反地，如果已经被限定在 Windows 平台上（比如，你已经使用了 DirectMuisc 的动态音乐功能），则花费时间去集成 OpenAL 是不必要的。如果你已经使用了 DirectMusic 的结构，则多半会坚持用 DirectX Audio 来实现 3D 音效。

PDG

第 15 章

3D 音效中的高级技术

前面两章学习了怎样应用两种不同的3D音效API实现基本的3D位置音效,但实现3D音效的技能还不止这些。

本章将关注3D音效中的一些高级技术,重点介绍如何应用环境建模(environmental modeling)技术来完善3D环境,另外还将简介两种建立3D环境的高级技术:环境音效扩展EAX(Creative's Environmental Audio Extensions)技术和杜比(Dolby)技术。

15.1 环境建模

3D音效除了源和听者之外,还有很多因素。前面的两章忽略了一个非常重要的变化因素:声音所发生的环境。例如,根据前两章所获得的知识,可以创建几可乱真的脚步声。为玩家创建并绑定一个听者,为每一只脚创建一个源,然后随着腿的移动发出声音——听起来没什么问题,但还是有一个因素被忽略了:在一个大型图书馆大理石地板上发出的脚步声与一个小厨房大理石地板上发出的脚步声肯定大不相同,图书馆中的大空间使得声波的传递与小厨房中的大不相同。

庆幸的是,有一些API可以模拟这些环境效应,并最终得到环境对声音影响的合理近似。如果播放footsteps_on_marble.wav,环境效应的处理将使得脚步声听起来像是从大型图书馆或者小厨房的大理石地板上发出来的。

15.1.1 用I3DL2实现混响效果

混响是环境的一个重要属性,这是无可争议的。在本小节中,学习怎样应用混响去反映一个特定的环境,以及怎样在游戏中应用I3DL2去实现混响效果。

(1)什么是I3DL2

I3DL2是第二级互动3D音频(interactive 3D audio, level 2)的缩写。I3DL2是由互动音频特殊兴趣组(interactive audio speacial interest group)提出的,这个组织由来自游戏研发及多媒体业界的专家组成,最早始于1994年,稍后成立了该工作组。1998年,这个工作组提出了IA-SIG互动3D音频暄染和评估指导性建议(interactive 3D audio rendering and evaluation guidelines)的第1级标准(Level 1)。1999年,对98建议作出改进后,发布了IA-

SIG 互动 3D 音频暄染指导性建议(interactive 3D audio rendering guidelines)的第 2 级标准(Level 2),即 I3DL2。

(2) I3DL2 相关术语

在学习如何使用 I3DL2 为环境建模编码之前,需要先了解一下关于混响的术语。在对环境进行建模前,必须理解听觉环境中影响回声和抑止的因素。

假设游戏中有一个大房间(如大教堂),其中有一个声源(如一个打呼噜的魔鬼)和一个听者(一位勇敢的英雄),二者之间间隔一定距离,如图 15.1 所示。

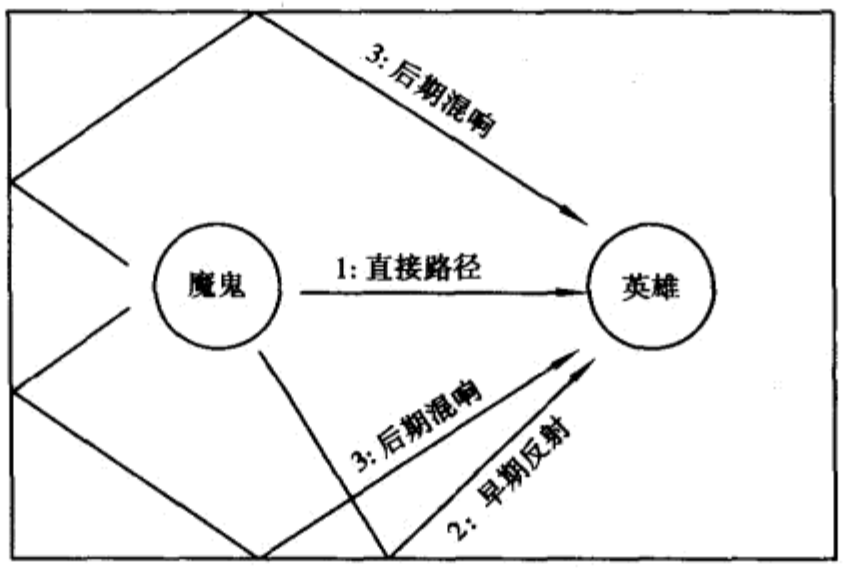


图 15.1 直接路径、早期反射和后期混响演示

当魔鬼打呼噜时,这位勇敢的英雄将会接受到几种不种类型的声波。首先,他会收到直接由魔鬼传过来的声波(路径 1),这称为直接声音。第二,他也会收到由相同的方向传来但在传递过程中由墙反射过来的声波(路径 2),这称为早期的反射声音。最后,英雄将会收到向相反方向传递,但经由多面墙体反射过来的声波(路径 3),这种类型的声波统称为后期的混响声音。图 15.2 画出了这 3 种不同声波的波形。

当然,这种描述十分简化,但基本的模型就是如此。使用 I3DL2 对环境进行建模,需要调整模型内的变量。例如,在 I3DL2 中,有一个特殊的变量控制听者听到从直接路径传来的声音与第一段早期反射声音之间的时间间隔。要模拟一个大房间的效果,需要增加此变量的值,因为在一个大房间里,声波沿着路径 2 到达一面墙之前需要传递更远的距离。要模拟一个小房间的效果,应使直接路径和早期反射间的时间间隔很小,甚至小到听者无法分辨出来。I3DL2 允许调整 3 条声波路径(直接路径、早期反射、后期混响)的延迟和音量。

同时,当声音反射时,高频声音与低频声音表现出差异,特别是高频声音会衰减得更快。而且,不同的房间以不同的散发度和密度反射声音。I3DL2 允许调整所有的这些变量,

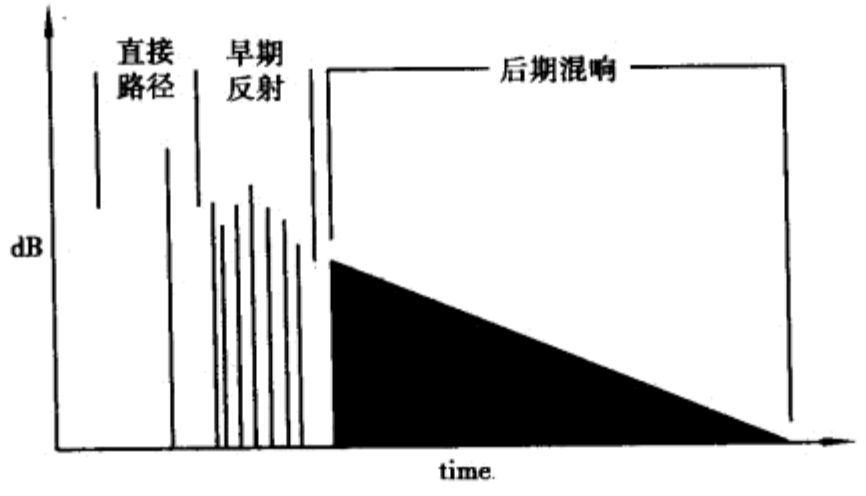


图 15.2　不同型声波的波形

以模仿广泛差异的听觉环境。

(3) I3DL2 是一种特效

在 DirectX Audio 里,I3DL2 被当作一种特效来处理,其特效接口为IDirectSoundFXI3DL2 Reverb8,这意味着你可以像使用其他特效一样使用它(见第 12 章中介绍的方法)。这使得我们使用 I3DL2 十分方便:将特效绑定到源上,设置它的参数即可,然后特效本身便会根据设定的参数去改变声音。

要将 I3DL2 特效集成到音频引擎中,需要建立一个新的特效类。下面是这个新类的声明描述:

```
class CI3DL2Effect : public CEffect
{
public:
   CI3DL2Effect() { ZeroMemory(&m_Params, sizeof(DSFXI3DL2Reverb)); ClearPreset(); }
   virtual ~CI3DL2Effect() { }
   virtual void SetAllParameters(IDirectMusicAudioPath8 * audiopath);
   virtual GUID GetGuid() { return(GUID_DSFX_STANDARD_I3DL2REVERB); }
   virtual void SetPreset(DWORD i3dl2preset);
   virtual void ClearPreset() { m_Preset = -1; }
   DSFXI3DL2Reverb m_Params;
protected:
   DWORD m_Preset;
};
```

注意,这个类除了具有与其他特效类相似的元素之外,还有一些新的函数和变量。这里有两个新的函数:SetPreset 和 ClearPreset,以及一个新的保护成员变量 m_Preset。在随后的预设效果一节中,会看到它们是如何工作的。

(4) I3DL2 混响参数

从现在开始,请仔细阅读 DSFXI3DL2Reverb 的结构,见表 15.1,它是 I3DL2 混响处理的核心。这个结构中的参数负责向 DirectX 特效描述应该如何修正声音。这里给出了每个参数的详细解释。注意,之所以不按照顺序解释,是因为一些变量的解释是以其他解释为基础的。

提示

注意,一个毫分贝(millibel,简写为 mB)是一分贝的百分之一,即一百个毫分贝等于一分贝。几个将在后面述及的参数描述的是以毫分贝为单位的衰减(attenuation)。如果指定 -50 mB 的衰减,即相当于指定半分贝的音量降低。

表 15.1 DSFXI3DL2Reverb 结构的成员变量

参数名称	描述
flHFReference	高频指示,以 Hertz 为单位。后面的几个参数与高频信号的处理有关。这个参数告知 DirectX 什么样的信号算是高频的。比如,将此值设定为默认的 5 000 Hz,就等于告知 DirectX 高于 5 000 Hz 的为高频信号
lRoom	房间衰减指数,以毫分贝(mB)为单位。这个参数影响房间效应的强度。记住,这是一个衰减指数,这意味着最高音量时对应指数为 0,任何低于最高音量的音量对应指数为负数。本参数的默认值为 -1 000 mB(-10 分贝)
lRoomHF	高频信号的房间衰减指数,以毫分贝(mB)为单位。在真实环境下,不同的房间对于高频信号的抑止效应与低频信号是不一样的。通过这个参数,可以告知 DirectX 给高频信号(频率高于 flHFReference 参数的信号)多大的衰减。例如,可以指定此参数为 -10 000 mB 而指定 lRoom 为 -1 000 mB,以创建一个完全屏蔽高频信号,仅允许低频信号通过的环境。这类似于在水下的效果

(续表)

参数名称	描述
flRoomRolloffFactor	房间流逝因子,用于告知 DirectX 将听者和源距离变远时,混响和反射效应以多快的速度消减。需要注意此参数只影响混响效应,源的流逝因子依然由听者控制
flDelayTime	回声持续的时长(以秒为单位)。如果将此值设定得很高,则声音听起来会像在大峡谷中一样(大量的回声)! 一个较小的值可以模拟一个压抑的房间,没有回声
flDecayHFRatio	高频消逝时间与低频消逝时间的比。如果将此值设为 1,则高频回声和低频回声将持续相同的时间。如果此值大于 1,则高频持续的时间更长(真实情况通常不是这样)。此参数的默认值为 0.83
lReflections	相对于 lRoom 的早期反射衰减指数,以 mB 为单位。早期反射是指听者在一个房间中听到的第一个回声(参见图 15.1)。这个参数告知 DirectX 是否早期反射的音量与 lRoom 相同,或者更强,或弱一些。此参数的默认值为 -2 602 mB,使得早期反射要柔和一些
flReflectionsDelay	直接路径和早期反射之间的时间间隔(以秒为单位)。简单地说,即是 DirectX 在声音播放后多少秒开始播放回声。此参数默认值为 0.007 秒。要模拟更大的房间,增加此参数的值,相反,减少此参数的值
lReverb	相对于 lRoom 的后期混响衰减指数,以 mB 为单位。与 lReflections 参数相似,此参数指出,后期混响衰减比 lRoom 强还是弱。此参数的默认值为 200 mB,比 lRoom 稍强一些
flReverbDelay	最早的早期反射和后期混响的时间差,以秒为单位。换句话说,此参数告知 DirectX 在播放第一段反射回声后要等待多长时间再播放后期混响
flDiffusion	后期混响延迟回声的密度,以百分比表示。默认值为 100%
flDensity	后期混响延迟模态密度(modal density),以百分比表示。默认值为 100%

在预定义的 DirectX 头文件中,所有的这些参数都有最大、最小及默认值。这些值遵循如下所示的命名规范:DSFX_I3DL2REVERB_(parameter)_MIN,DSFX_I3DL2REVERB_(parameter)_MAX 及 DSFX_I3DL2REVERB_(parameter)_DEFAULT。例如,lRoom 的 3 个预定义值为 DSFX_I3DL2REVERB_ROOM_MIN,DSFX_I3DL2REVERB_ROOM_MAX 及 DSFX_I3DL2REVERB_ROOM_DEFAULT。

(5)预设效果

要学会怎么调整这些不同的参数以实现一个听起来不错的环境(即准确地传递所期望的空间感觉),需要一些练习。简易起见,DirectX 有几个预设效果可供使用。它们覆盖广泛种类的环境,把读者从调整单个变量的工作中解放出来。

使用这些预设效果相当容易,只需调用 IDirectSoundFXI3DL2Reverb8 的 SetPreset 函数,并将对应的 GUID 值传递给它。完整的 GUID 的列表可以在 DirectX 文档中关于 SetPreset 函数的部分里找到。每个预设效果的真实值是在 dsound. h 中定义的。

15.1.2 创建环境编辑器

正如读者所看到的,这些数字凭直觉无法知晓其意义。要正确地创建一个使用 I3DL2 混响的音频环境,需要一段程序帮助你调整这些参数并实时地听到这些参数所引起的混响效果。换个说法,需要一个环境编辑器。

好在没有多少变量,对于一个基本的环境编辑器来讲,设置有 12 个轨迹条的对话框就够用了,本节将介绍如何创建这个编辑器。图 15.3 是这个编辑器的一个界面,这个对话框有一堆的轨迹条,一个装载(Load)和保存(Save)按钮,以及浏览和播放 WAV 文件的按钮。

在分析 Ch15p2_I3DL2EnvEditor 示例程序之前,建议先运行一下这个程序。如果没有 Win32 GUI 程序设计的经验,这个示例也可以教你几招。

(1)初始化

在 Win32 程序设计中,要使用轨迹条(trackbar),需调用 Win32 API 的 InitCommonControls 函数。否则,将无法创建带有轨迹条的窗口。与其他控件(按钮、编辑框等)不同,轨迹条不属于 Windows GUI 原始分发版本的一部分。它是在后续过程中添加上去的,所以需要调用那个函数。

(2)使用轨迹条

这里,不打算描述创建一个无模式对话框及其消息循环的内容,许多介绍 Windows 编

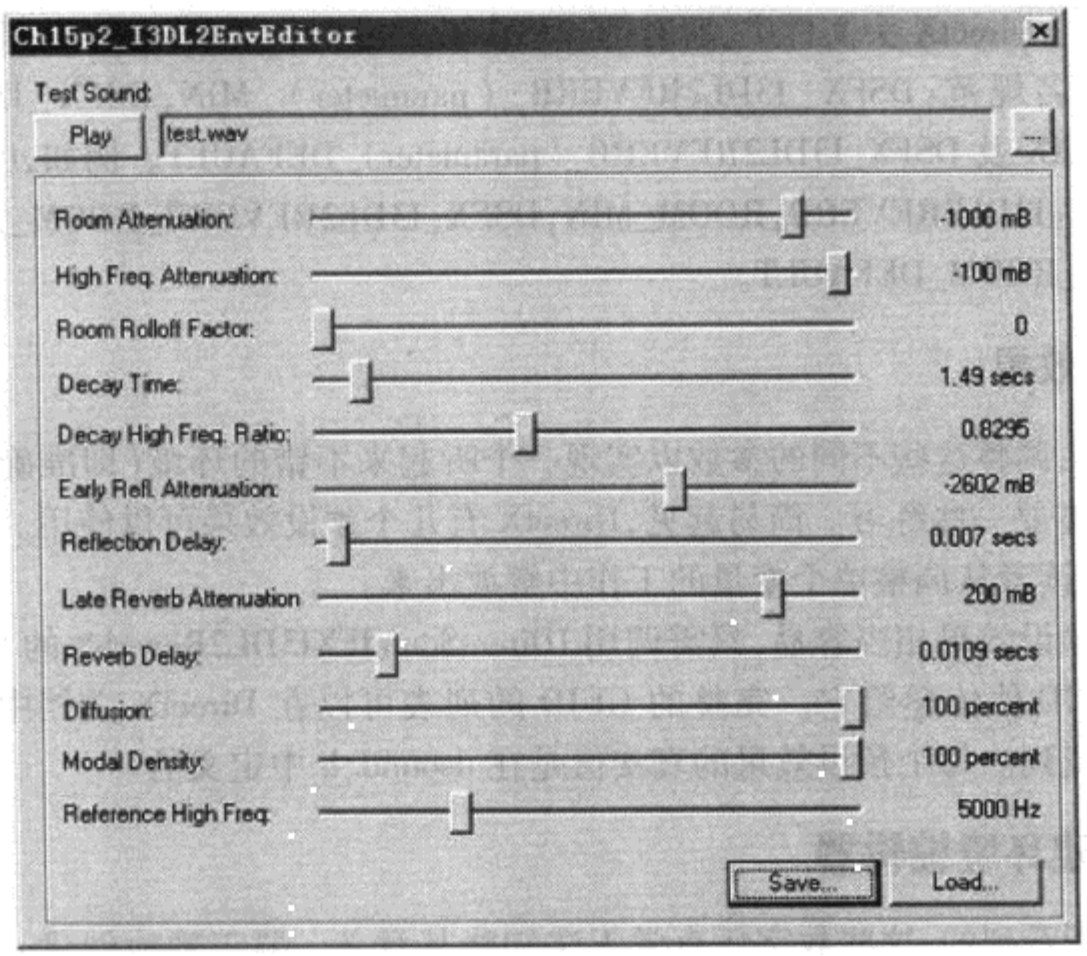


图 15.3 Ch15p2_I 3DL 2EnvEdit 示例程序的截图

程的书都包含了这些内容。本书只花少许笔墨介绍它们的功能是如何实现的。

与其他 Windows 控件一样,轨迹条也是靠 Windows 消息驱动的。表 15.2 列出了 3 个最重要的轨迹条消息。

表 15.2 常用轨迹条消息

轨迹条消息	含义
TBM_SETRANGE	设置轨迹条的最小和最大值
TBM_SETPOS	设置轨迹条的当前位置
TBM_GETPOS	返回轨迹条的当前位置。当用户移动轨迹条的位置时,轨迹条将通过向其父窗口发送 WM_HSCROLL 消息(横向轨迹条)或 WM_VSCROLL 消息(纵向轨迹条)以通知变化

在初始化对话框时,需要做的事情就是向轨迹条传递 TBM_SETRANGE 和 TBM_SETPOS 消息;然后,搜寻 WM_HSCROLL 消息;最后,向轨迹条发送 TBM_GETPOS 以获取其新位置。

特别注意,轨迹条仅接受整数。对大多数应用而言,这不是问题,但本示例程序中有些参数必须为浮点数。例如,混响延迟变量 flReverbDelay,最小值 0.0,最大值 0.1。

> **提示**
>
> 这里所描述的仅是轨道条的一部分功能,还有更多没有描述。比如,你可以给轨道条加上刻度,可以让它与其他控件相关联。要获得更多信息,请参见位于 Platform SDK\User Interface Services\Shell and Common Controls\Trackbar Controls 中的 MSDN 文档。

解决这个问题,必须扩大轨迹条的最大值。例如,你也许会选择给 flReverbDelay 乘 10 000,以得到从 0 ~ 1 000 的整数区间。这样就没问题了,只是需要记住:在将 flReverbDelay 的值反应到轨迹条上时要乘以 10 000,而使用 TBM_GETPOS 取数时要除以10 000。

在 Ch15p2_I3DL2EnvEditor 示例程序里,有两个名为 RefreshSlider 的重载函数,一个接受整数参数,另外一个接受浮点参数和一个乘数因子。

(3)装载和保存环境

装载和保存一个结构的最好方法就是直接将其内存镜像作为一个二进制文件保存。本示例程序就是这么做的,以下是装载和保存环境的函数:

```
void OnLoad(HWND hDlg)
{
  char filename[MAX_PATH] = {0};
  OPENFILENAME ofn = {sizeof(OPENFILENAME), hDlg, NULL,
                      "Reverb Files\0*.reverb\0All Files\0*.*\0\0", NULL,
                      0, 1, filename, MAX_PATH, NULL, 0, NULL,
                      "Open Reverb File",
                      OFN_FILEMUSTEXIST|OFN_HIDEREADONLY, 0, 0,
                      TEXT(".reverb"), 0, NULL, NULL};
  // 显示 OpenFileName 对话框。然后,装载指定文件
  if (GetOpenFileName(&ofn))
  {
    int handle = open(filename, O_RDONLY | O_BINARY);
```

```
        if (handle == -1) Throw("Cant open file!");
        read(handle, &g_ReverbValues, sizeof(DSFXI3DL2Reverb));
        close(handle);
        RefreshGUI(hDlg);
    }
}
void OnSave(HWND hDlg)
{
    char filename[MAX_PATH] = {0};
    OPENFILENAME ofn = {sizeof(OPENFILENAME), hDlg, NULL,
                        "Reverb Files\0*.reverb\0All Files\0*.*\0\0", NULL,
                        0, 1, filename, MAX_PATH, NULL, 0, NULL,
                        "Save Reverb File",
                        OFN_HIDEREADONLY, 0, 0,
                        TEXT(".reverb"), 0, NULL, NULL};
    // 显示 OpenFileName 对话框。然后,装载指定文件
    if(GetOpenFileName(&ofn))
    {
        int handle = open(filename, O_RDWR | O_BINARY | O_CREAT | O_TRUNC);
        if (handle == -1) Throw("Cant save file!");
        write(handle, &g_ReverbValues, sizeof(DSFXI3DL2Reverb));
        close(handle);
    }
}
```

这段代码使用了常用的文件打开和文件保存对话框。要使用这些对话框,需要先准备一个 OPENFILENAME 结构。可以在 MSDN 中查询 GetOpenFileName 函数的说明,以弄清 OPENFILENAME 结构中的每个成员变量的含义。

准备好 OPENFILENAME 结构后,调用 GetOpenFileName 函数即可显示对话框。如果用户点击了对话框的 OK 按钮,则函数返回 true,前述的代码将会自动装载或保存文件。

(4)完善编辑器

这里给出的仅仅是一个简单的环境编辑器。专业游戏的环境编辑器要比这复杂得多,它需要考虑玩家在游戏世界中的位置,而且还必须允许音频设计者为不同游戏级别的不同

场景设定不同的环境参数。

本书提供一些建议,可以尝试提升程序的如下几个方面:

- 用更标准的格式去存储参数,如 XML。
- 增加对预设效果(preset)的支持。设想一下,可以通过下拉菜单选择预设效果,然后做一些修改,再保存为新的环境。
- 除编辑环境参数外,还允许用户编辑 DirectSound 3D 听者及源的属性。这种事情很频繁,对一个对象的修改通常会导致对另一个对象的调整。

15.1.3 I3DL2 的其他功能

I3DL2 的功能不仅仅只用于混响。I3DL2 同样规定了关于阻隔(occlusion)及障碍(obstraction)的接口,这两个概念对于 3D 音效而言同样重要(详见后面的定义)。通过 DirectSound 属性扩展,就可以使用阻隔和障碍的功能,使你的游戏音效更加逼真。

要想获得关于 I3DL2 的更多信息,请参见位于 http://www.iasig.org 的 IA-SIG3D 音效工作组网站。

15.2 环境音效扩展技术 EAX

这里,再介绍一些其他技术。如果读者希望站在 3D 音效的前沿,这些技术将非常有用。首先要介绍的一项环境音效扩展技术称之为 EAX。

EAX 是 Environmental Audio Extensions 的缩写。这项技术由 Creative Labs 创立,并经历了两个版本(到本书写作时,EAX2.0 是最新的)。要理解 EAX,必须先了解在一个环境里,声音是如何因环境影响而改变的,这包括如下几个概念:

- **混响**:读者已经学过,混响通过数学方法模拟声波在墙等障碍物上的反射,以创造更加真实的回声效果,并最终模拟不同的环境。
- **隔阻和障碍**:隔阻意味着物理上将听者和声源隔开(如一面墙)。而障碍是指听者和声源处于同一个空间中,但中间有一些障碍物(比如一个柳条箱)。
- **声音方向**:在现实生活中,并非所有频率的声音都会沿你所确定的方向均匀分布。低频信号在传播方向的发散性上与高频信号不同。我们设置家庭影院时,低频扬声器置于何处并不重要,但中频及高频扬声器的位置和方向却很要紧,原因就在这里。高级 3D 音效的 API 允许你独立地调整低频及高频信号的声音方向。

这是环境建模的 3 个核心概念。而 EAX 将这 3 个理念付诸实施。

DirectSound 及 OpenAL 都支持 EAX,而且 EAX 在两种工具中的使用都很简单。在 DirectSound 里,只需简单地使用绑定到一个源上的 EAX 对象的接口;对于 OpenAL,EAX 是

一个扩展,通过 OpenAL 的扩展机制,使用它也非常方便。

访问 Creative 的开发者的 Web 站点,可以获取更多关于 EAX 的信息,地址为 http://developer. creative. com。

15.3 杜比数字编码技术

如果读者是专业游戏的专业开发者,建议关注一下杜比编码,特别是杜比数字编码(Dolby Digital Encoding)。这种编码技术可以将 5 个离散的声道编码成为一个单一的比特流。以前我们只能将两个声道编码给一个立体声 WAV 文件,而通过杜比编码,我们可以让左面、右面、中间、左周围(left surround)、右周围(right surround)5 个声道编码为一个杜比比特流。这个比特流同时还支持低频效应 LFE(Low-Frequency Effects)、声道,以包含那些低于 120 Hz 的声音。

要获取更多关于杜比数字编码技术的信息,请访问 http://www. dolby. com。

15.4 本章小结

本章介绍了 3D 音效中的一些高级技术。现在,已经具备了为游戏设计 3D 音效并集成那些诸如 EAX 的高级技术。

3D 音效的未来十分光明。市场上已经出现了可以用硬件模拟单个声波反射的声卡。摩尔定律正在继续将专业级的音频设备带到普通消费者手中,每一代新的声卡都会具备更多的功能以支持更好的声效。如果读者曾认为“更好的游戏音效就是更多的声道和更高的采样频率”的话,希望本章已经改变了这个认识。

第 16 章

DirectPlay 语音

本章的惟一目的就是教会读者如何使用 DirectPlay 的语音 API。这个少为人知的 DirectPlay 组件可以在多玩家游戏中帮助玩家进行实时语音交谈。这是一项相对容易实施的技术,它可以为游戏增添新的色彩。

16.1 DirectPlay 语音基础

设计 DirectPlay 语音 API 基于一个简单的理由:在多玩家游戏中如果能听到其他玩家的尖叫声将会非常有趣。例如伏击一个玩家,当他意识到落入你手中并大声诅咒时,没有什么比能亲耳听到他的诅咒声更令人满足了。有了 DirectPlay 语音接口,通过少数几行代码就可以在游戏中加入这种令玩家沉醉的功能。

16.1.1 基本概念

DirectPlay 语音通过 DirectPlay 连接传递语音数据。它是从录制玩家的语音开始的。启动语音录制可以手工完成(比如,玩家按一下某个键或者游戏杆的某个按钮以开始录音),也可以自动完成(当计算机检查到麦克风显著的音量变化时开始录音)。在简单应用模式下,当玩家讲话时,DirectPlay Voice 对麦克风进行采样(低采样频率),接着用一个专门设计的语音压缩算法对数据进行压缩。然后再通过 DirectPlay 通信协议将压缩后的数据发送到另一台计算机。接下来另外一台计算机对数据进行解压缩,再通过声卡播放出来。在理想状态下,这些处理必须很快,以使得语音不要断断续续,但只要是通过网络传输,说话声音和听者听到声音之间总会有一点延迟。

16.1.2 DirectPlay 语音传输的拓扑结构

刚才所描述的是点到点语音通信的例子。在这种拓扑下,每台计算机都必须把玩家的语音数据发送给其他所有计算机。这意味着如果同5 个人一起玩游戏,当自己说话时,计算机必须把语音数据发送给另外5 台计算机,如图 16.1 所示。

完全可以想象,这会堵塞上行和下行信道。标准的压缩语音需要占用 8 kb/s 的带宽(虽然可采用其他压缩编码,占用带宽从 1.2 ~ 13 kb/s)。如果有 5 个玩家,这意味着需要 40 kb/s(8 kb/s ×5)的上行带宽,以及 40 kb/s 的下行带宽(因为每个玩家都有可能说话)。

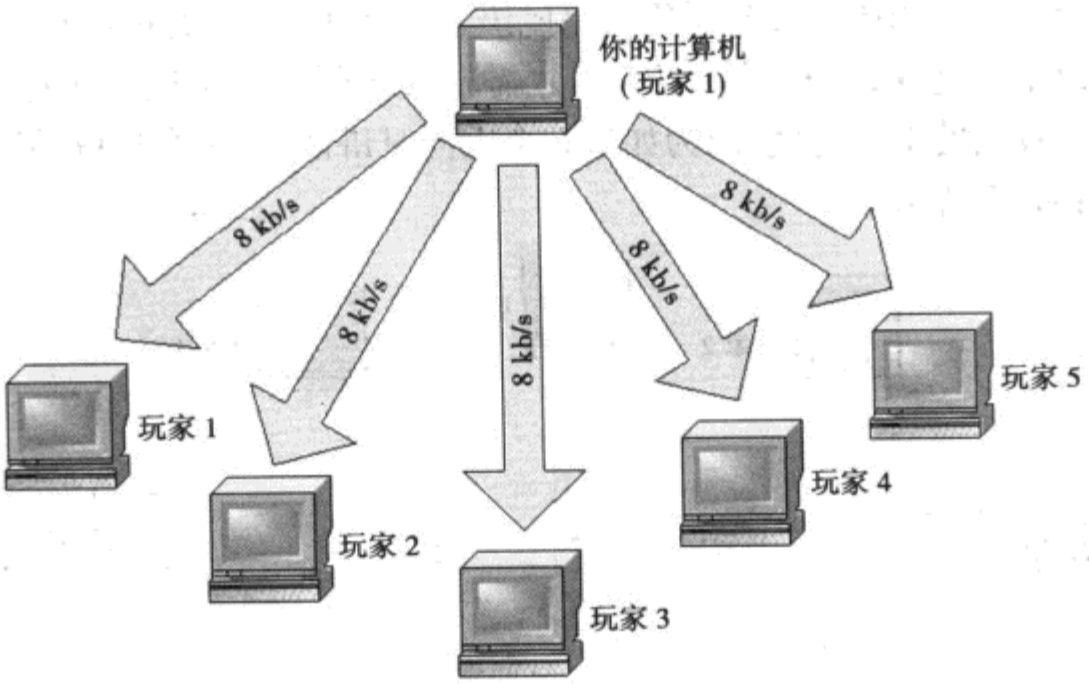


图 16.1 点到点通信环境里,有 6 个玩家时的语音传送情况

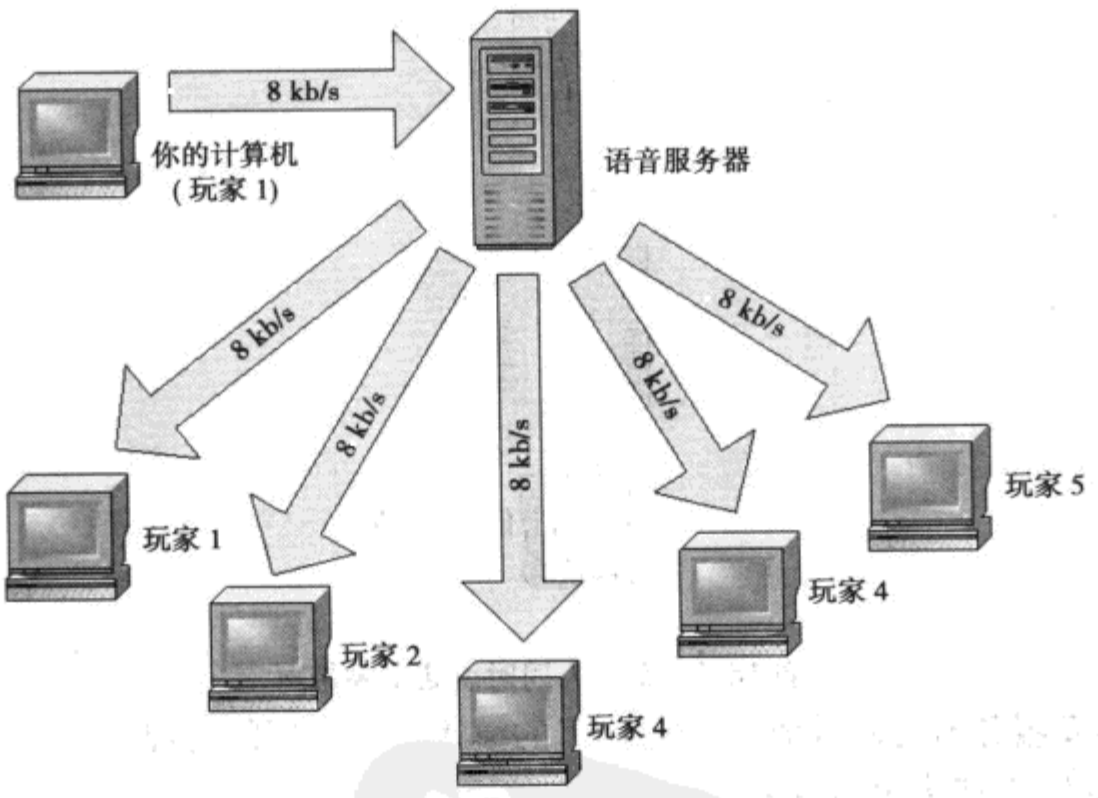


图 16.2 在前向服务器拓扑模式下,语音数据发送的情况

在局域网环境里,这没有问题,因为局域网具有最低 10 Mb 的带宽。但如果你通过 Modem 或者宽带上网,同时上下行的 40 kb/s 速度是难于达到的。

为此,DirectPlay 语音提供了另外两种拓扑:转发服务器拓扑(Forwarding Server Topology)和混音服务器拓扑(Mixing Server Topology)。

在转发服务器拓扑模式下,玩家说话时,其计算机将语音数据发送给一个中央语音服务器。然后再由服务器将语音发送给游戏中的其他计算机(如图 16.2 所示)。

这种方式由一台专用语音服务器代为完成语音发送工作，从而将计算机从发送多份数据拷贝中解放出来。这样，只有语音服务器才需要较快的上行速度。注意，这种模式下的客户机依然需要较快的下行速度，因为如果5个人同时讲话，一个客户机必须同时接受5份语音数据，如图16.3所示。

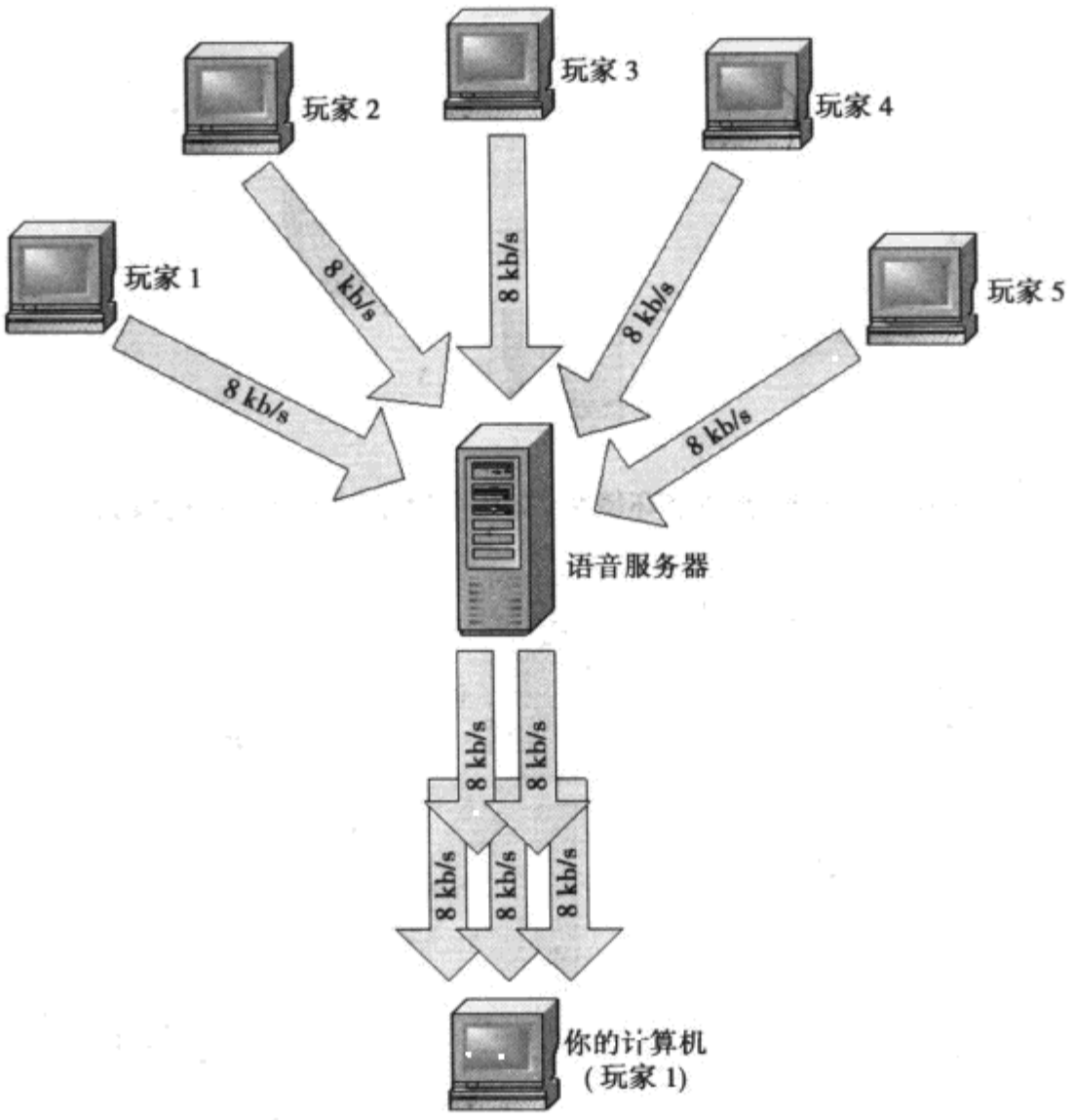


图16.3 5个人同时讲话所需带宽

可以把转发服务器技术看成“一上多下”的模式。在最坏的情况下，一个客户端需上传一份语音流，同时下载多份语音流。

第三种语音发送的拓扑结构称混音服务器拓扑，它是一种“一上一下”的模式。在这种拓扑下，服务器从所有的客户端接收语音数据，然后将数据解压，并为每一个客户端单独混制一个合适的语音流然后发送给它们，如图16.4所示。

例如，如果玩家1和玩家2同时讲话，服务器将会把玩家1的语音流发送给玩家2，把玩家2的语音流发送给玩家1，把二者混合的语音流发送给所有其他玩家。这样，一个客户端只需下载一个数据流。然而，服务器的工作量增大了，它需要为每一个客户端解压、混音、再压缩，而不是简单地像前向服务器拓扑一样转发语音数据。混音服务器拓扑牺牲服

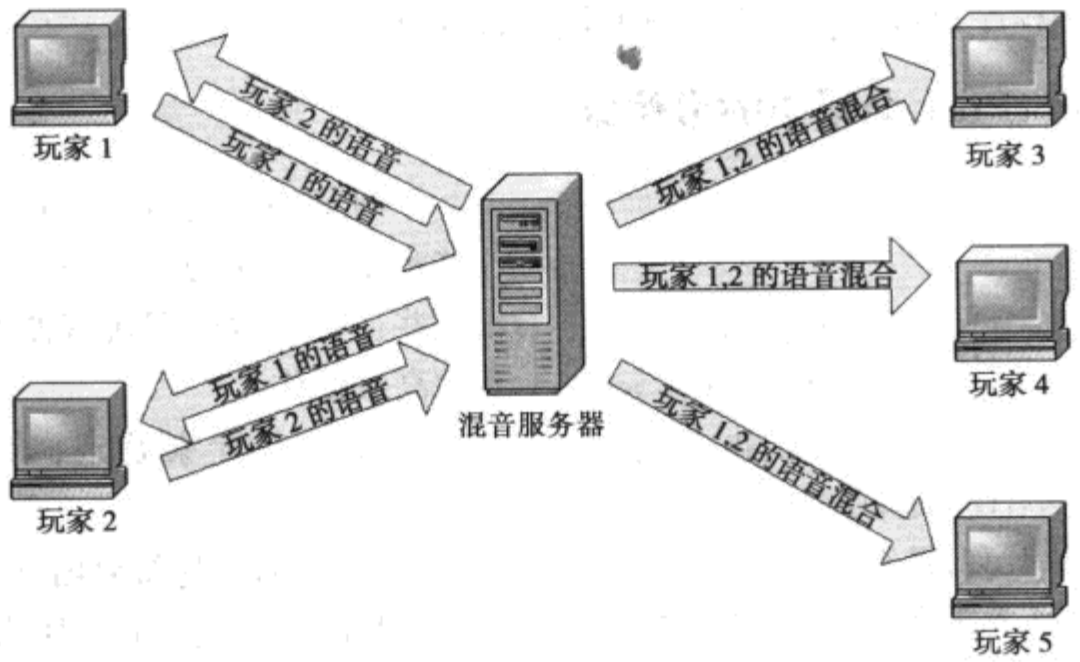


图 16.4 混音服务器为每个客户端混制合适的语音流

务器处理能力以换取较小的带宽占用。

至于哪种拓扑更好,这取决于游戏运行的方式。如果游戏在局域网上运行,因为有足够的带宽,那么应当选择点到点模式,这种方式可以保证每台计算机完成相同数量的工作。如果游戏基于 Internet 运行,则选取何种拓扑取决于是否有合适的服务器。混音服务器要处理大量的运算,对于大多数游戏而言,要让混音服务器再承担运行游戏本身的任务是不可能的,因为帧的速度会变得很慢。如果语音服务器是一台普通的玩家的计算机,那么转发服务器拓扑是最好的选择。然而,如果有单独的专用语音服务器,它除了网络传输之外没有别的负担,主要负责完成混音的任务,那么混音服务器拓扑是最好的选择。

提示

你可以书写代码让游戏自动决定选择何种拓扑结构(根据玩家所指定的网络设置),而不是让玩家去选择拓扑方式。举个例子,如果有客户端是通过 Modem 玩游戏的,则你的代码应自动排除使用点到点拓扑的可能性。

16.2 DirectPlay 速成

DirectPlay 语音是在 DirectPlay 之上运行的。这意味着如果读者从来没有 DirectPlay 编码的经验,那么在使用 DirectPlay 语音之前,需要先了解 DirectPlay 的基本原理。

本书不详细讲述 DirectPlay,只是快速浏览其基本层面的内容。了解 DirectPlay 的工作原理是理解 DirectPlay 语音如何工作的前提。所以,如果读者无法从简短的描述中理解 DirectPlay,建议最好学习一下关于 DirectPlay 更多的知识(比如,大厅及玩家组)。请查看

DirectPlay 的示例程序或者阅读关于 DirectPlay 的其他资源。

16.2.1 DirectPlay 会话中的事件流

> **提示**
>
> 不同的游戏使用不同的方法,以达到使用尽可能短小的消息传递信息的目的。绝大多数游戏会在游戏世界开始时进行一次状态的同步,然后仅发送和接收那些包含世界改变的信息(比如,服务器也许会发信息给客户机说:"4 号坦克刚才移动到了(5,16)位置")。

对某些人而言,理解 DirectPlay 有一些难度。解决方法是把过程拆开,关注那些连接和运行多玩家游戏时需要完成的高级操作。

对于多玩家游戏,有 3 个基本的步骤。

首先,需要连接到游戏。

其次,在运行过程中,游戏代码会不停发送和接收关于游戏状态的信息,比如,物体在 RTS 中的位置,玩家刚刚开火等等。游戏代码的大部分时间都花费在发送和接收消息上了。此外,还需要处理自定义的消息,并处理 DirectPlay 的系统消息(玩家加入,玩家断开等等)。

最后,是必须的"反初始化和清理"的过程。这是很重要的一步。因为如果忽略了这一步,当游戏中任一玩家退出时,DirectPlay将不会把这一消息通知其他玩家。

现在读者应对 DirectPlay 有了概略性的认识,为了了解得更仔细些,后续的章节将会讲解使用 DirectPlay 的细节。

> **提示**
>
> DirectPlay 同 DirectMusic 一样,也大量使用 GUID(全局惟一标识符)。在多玩家游戏中,DirecPlay 使用 GUID 以标识游戏、服务提供者和其他的一些事物。一个 GUID 值在时间和空间上都是惟一的,你可以放心它不会同已有的 GUID 值重复,也不会同将来创建的 GUID 值重复。

16.2.2 创建 DirectPlay 接口

需要完成的第一件工作是创建 DirectPlay COM 对象的接口。这个接口根据游戏所使用的拓扑结构的不同而不同:点到点、或客户/服务器结构。表 16.1 给出了所有可用接口。

本书的示例程序采用的是点到点拓扑,故使用 IDirectPlay8Peer 接口。下面的描述将围绕此接口展开。事实上,除了连接过程不同外,3 个接口拥有几乎一致的方法集合,因此使用它们时几乎大部分东西都是一样的。

表 16.1 可用的 DirectPlay 接口

接 口	描 述
IDirectPlay8Peer	在点到点拓扑下使用的接口
IDirectPlay8Client	在客户/服务器模式中使用此接口作为客户连接到拓扑结构中
IDirectPlay8Server	使用此接口创建一个新的服务器或者新的客户/服务器会话

16.2.3 选择服务提供者

当为网络拓扑创建好合适的接口后,需要指定一个服务提供者。DirectPlay 可以工作于多种不同的网络连接之上(TCP/IP、以及 Modem 和串行连接)。每个连接具有不同的服务提供者。应该从 DirectPlay 获得它所支持的服务提供者的列表,并展现给用户,由用户来选择使用哪一个服务提供者。

通过调用选定的 DirectPlay 接口的 EnumServiceProviders 函数,可以列出所有可用的服务提供者。

16.2.4 连接到一个游戏

前面的工作已经将系统支持的所有可用的服务提供者显示给用户,用户也作出了选择。在一个点到点游戏世界里,下一步由玩家选择:是连接到一个已有的游戏,还是创建一个新游戏。

通过调用 IDirectPlay8Peer 的 EnumHosts 可以得到可用的游戏主机的列表(游戏主机指那些运行着游戏会话的计算机)。

读者也许会感到疑惑,DirectPlay 如何知道一台计算机是否运行着游戏呢? 这得益于 GUID。每个 DirectPlay 应用都有自己的 GUID 值,对于内部运行的 DirectPlay,GUID 即是游戏的名字。在调用 EnumHosts 时,可以想象计算机在向局域网所有计算机广播以下问题:“如果你运行着这个 GUID 所标示的游戏,请答复”。接着计算机便收集所有的回复并反馈给代码。在这一过程中,每一个运行着对应游戏的主机都会返回一个 GUID 值,即是说:“这台主机正在运行你所感兴趣的游戏”。

接下来将主机列表展示给用户,由用户选择主机。然后通过调用 IDirectPlay8Peer 接口的 Connect 函数将选定主机的 GUID 值传递给 DirectPlay,DirectPlay 将返回成功或者失败的信息。如果连接成功,便可以开始发送和接收消息了。

如果用户选建一个新游戏而不是连接到一个已存在的游戏,则过程要简单一些,只需要调用 IDirectPlay8Peer 接口的 Host 函数,并向其传递一个 DPN_APPLICATION_DESC 结构。这个结构包含了希望创建的新游戏的信息。比如,这个特殊游戏的名称、密码(如果游戏是保密的)、以及最大玩家数等。如果 Host 函数返回值为成功,则可以开始发送和接收消息了。

表 16.2 常用 DirectPlay 消息

消息名称	描 述
DPN_MSGID_CREATE_PLAYER	当新玩家加入到游戏中时,DirectPlay 将向你的代码发送此消息。当你收到消息后,你可以调用 IDirectPlay8Peer 接口的 GeiPeerInfo 方法查询玩家的名称和其他信息,然后使用这些信息去初始化游戏特定的玩家数据
DPN_MSGID_DESTROY_PLAYER	当一个玩家断开游戏连接时,DirectPlay 将向你的代码发送此消息。你的工作是要确保在你的游戏代码中已清除了关于此玩家的链接
DPN_MSGID_RECEIVE	当接收到来自于其他计算机的特定游戏时,DirectPlay 将发送此消息给你的代码。消息中包含一个指向接收到的数据的指针;你的工作是按照期望的格式解析这些数据,并做出相应的处理
DPN_MSGID_SEND_COMPLETE	当要发送的数据在事实上已经从物理链路上发送出去以后,DirectPlay 会向你的代码发送此消息。在 DirectPlay里,数据的发送是异步的,即作出发送请求后,发送过程将会在后台处理,同时游戏将接着完成其他的工作

16.2.5 接收消息

在多玩家游戏中,来来往往的信息都是以自定义的格式传递的。然而,也存在一些必须做出反应的 DirectPlay 预定义消息。这些消息的完整列表在这里没有给出(在 DirectPlay 文档里),表 16.2 列出了基本的消息集合。

这是 DirectPlay 里最常用的 4 个消息。

在初始化 IDirectPlay8Peer 接口时,有一个参数必须提供,这个参数是一个指向回调函数的指针,当有消息进来时,DirectPlay 会调用此函数。从这个角度看,对 DirectPlay 编程有

点类似于 Windows 对话框编程。回调函数将会接收到作为参数提供的消息的 ID 值,及指向消息缓冲区的指针。

在回调函数中,你的工作是检查收到的消息并作出合适的反应。通常,这些工作是通过一个 swtich 语句完成的,代码如下所示:

```
HRESULT WINAPI OnDirectPlayMessage( PVOID pvUserContext, DWORD dwMessageId,
PVOID pMsgBuffer )
{
  switch( dwMessageId ) {
    case DPN_MSGID_CREATE_PLAYER:
      {
        // 收到一个创建玩家消息!
        HRESULT hr;
        PDPNMSG_CREATE_PLAYER pCreatePlayerMsg;
        pCreatePlayerMsg = reinterpret_cast < PDPNMSG_CREATE_PLAYER >
        (pMsgBuffer);
        DWORD dwSize = 0;
        DPN_PLAYER_INFO * pdpPlayerInfo = NULL;
        // 在这里完成其他工作……
      }
      break;
    // 忽略其他 case 语句
  }
Return(S_OK);
}
```

如果回调函数正确处理了消息,则返回 S_OK。也可以返回其他值,作为对某些特定消息的反馈,这些消息在 DirectX 文档中有描述。

16.2.6 发送消息

在多玩家游戏中,可以通过调用 IDirectPlay8Peer 接口的 SendTo 函数将数据发送给其他的计算机。函数以参数形式接受所期望发送的玩家的 ID 号(或者其他预定义值,如 DPNID_ALL_PLAYERS_GROUP 表示所有玩家),送给一个 DPN_BUFFER_DESC 结构,此结构中包含一个指向待发送数据的指针,以及待发送数据的字节总数。同时,SendTo 方法还接收一些标志性参数,以便更好地调整其行为。比如,可以指定 DPNSEND_GUARANTEED

标志,让 DirectPlay 保证消息正确传递到目的地。

16.2.7 线程与同步

DirectPlay 程序设计区别于其他部分的最重要的因素在于它是一个多线程组件。注意:一定要记住 DirectPlay 可能会从多个不同的线程同时调用回调函数。

这意味着必须确保回调函数的线程是安全的。如果要修改全局数据(比如,不是在堆栈,而是全局定义的数据),这些修改代码必须放在一个临界区中,或者必须处于其他线程同步原语的保护之下。

16.2.8 断开

当玩家决定退出游戏时,一定要记住调用 IDirectPlay8Peer 接口的 Close 函数。这个函数将通知每个人此玩家已退出的消息。同时代码也要调用大厅(lobby)接口的 Close 函数,这个接口名为 IDirectPlay8LobbiedApplication。代码关闭了两个接口之后,就可以调用 Release 或者使用 SAFE_RELEASE 宏释放它们以及其他的 COM 接口。

16.3 使用向导 CNetConnectWizard

完成前 3 步需要大量代码。作为 DirectX SDK 的一部分,微软提供了一个向导类以帮助简化代码,这使得在应用中可以方便地设置 DirectPlay 并连接到一个游戏。

这个类名为 CNetConnectWizard,可以在 DirectX 安装目录的 Samples 子目录中找到。它可以帮助用户轻易地选择服务提供者,连接到一个游戏或者开始一个新游戏。惟一的缺点是此类通过一些平常的对话框来完成这一系列动作,但游戏设计者却通常希望使用自定义的 GUI,有漂亮的按钮和滚动条。这个向导无法简单地完成这些工作,所以大多数时候,需要自行书写代码。但如果不介意在游戏中显示一些通常的对话框,可以使用 CNetConnectWizard 以节省时间。

16.3.1 创建和初始化向导

CNetConnectWizard 通常以动态方式创建,如下述代码所示(此段代码摘自 Ch16p1_VoiceChat):

```
CNetConnectWizard * g_pNetConnectWizard = NULL;     // 连接向导
g_pNetConnectWizard = new CNetConnectWizard( hInstance, NULL,
    "Ch16p1_VoiceChat", &g_GameGUID );
g_pNetConnectWizard -> Init( g_pDPPeer, g_pDPLobbiedApp );
```

该代码创建了一个新的 CNetConnectWizard 实例并调用其 Init 函数。此 Init 函数接受两个接口参数:IDirectPlay8Peer 及 IDirectPlay8LobbiedApplication。

本书前面故意忽略了对大厅(lobby)及 IDirectPlay8LobbiedApplication 的解释。大厅是 DirectPlay 的一项高级特性,只有当希望游戏程序可以在一个游戏选择器(大厅)里启动时才需要用到大厅功能。可以把大厅理解成一种从一个程序向另一个程序传递连接和游戏设置的机制。如果感兴趣,可以阅读 DirectX 文档获取更多信息,但对于本章,只需知道 CNetConnectWizard 需要一个大厅接口就够了。

16.3.2 绑定回调函数

只有当 CNetConnectWizard 能够处理 DirectPlay 消息时,它才可以发挥作用。这意味着 DirectPlay 回调函数必须调用 CNetConnectWizard 的 MessageHandler 函数,并向其传递相同的参数。

与 IDirectPlay8Peer 接口一样,IDirectPlay8LobbiedApplication 接口也需要一个回调函数,而且 CNetConnectWizard 必须以同样的方法绑定这个回调函数。两个绑定的过程如下述代码所示(该代码摘自 Ch16p1_VoiceChat):

```
HRESULT WINAPI OnDirectPlayMessage( PVOID pvUserContext, DWORD dwMessageId,
PVOID pMsgBuffer )
{
  switch( dwMessageId ) {
    case DPN_MSGID_CREATE_PLAYER:
      // 省略
      break;
    case DPN_MSGID_DESTROY_PLAYER:
      // 省略
      break;
      // 其他消息
}
// 让向导有机会对消息作出反应
return g_pNetConnectWizard -> MessageHandler(
      pvUserContext, dwMessageId, pMsgBuffer );
}
HRESULT WINAPI OnDirectPlayLobbyMessage(
      PVOID pvUserContext, DWORD dwMessageId, PVOID pMsgBuffer )
```

```
{
    return g_pNetConnectWizard -> LobbyMessageHandler(
        pvUserContext, dwMessageId, pMsgBuffer );
}
```

IDirectPlay8LobbiedApplication 接口的回调函数 OnDirectPlayLobbyMessage 并没有完成什么具体的工作,它只是把获得的参数传递给向导。然而,IDirectPlay8Peer 的回调函数认真观察消息类型并处理它们,然后调用向导的 MeesageHandler 函数,使其有机会处理消息并返回值。

16.3.3 设置默认值并启动向导

当创建好 CNetConnectWizard 并将其绑定到 IDirectPlay8Peer 及IDirectPlay8LobbiedApplication 的回调函数之后,便可以设置其默认值,然后显示对话框供用户选择。

下述代码显示了这一过程:

```
// 设置默认值
g_pNetConnectWizard -> SetPlayerName( "Anonymous" );
g_pNetConnectWizard -> SetSessionName( "New Game" );
g_pNetConnectWizard -> SetPreferredProvider( "DirectPlay8 TCP/IP Service Provider" );
// 显示连接向导
hr = g_pNetConnectWizard -> DoConnectWizard( FALSE );
if (FAILED(hr)) throw(hr, "Connection wizard failed.");
if( hr == NCW_S_QUIT )  { return(0); }
```

代码调用向导的 SetPlayName、SetSessionName、SetPreferredProvider 函数,设置其默认值。这些默认值将显示在提供给用户的对话框中,如图 16.5 及 16.6 所示。

在 DoConnectWizard 函数被调用时,向导将会显示对话框让用户选择。DoConnectWizard 函数返回一个 HRESULT 值,如果该值显示失败,则意味着向导出现了错误。如果HRESULT 值为 NCW_S_QUIT,这意味着用户在向导结束之前选择退出并关闭向导。这通常表示用户希望退出多玩家游戏,此时,应该让其退出,并显示游戏标题界面,或者直接退出程序,Ch16p1_VoiceChat 示例程序就是这样处理的。

如果用户走完了向导的所有过程,则向导最后会调用 IDirectPlay8 接口的 Host 或者 Connect 函数,完成相应的工作。

可以保存用户的选择以作为下次向导的默认值。可以通过调用 GetPlayName、GetSessionName 及 GetPreferredProvider 获取这些默认值,然后再将其保存在注册表或者配置文件中,下次显示对话框前再将其读回。同时,也可以调用 IsHostPlayer 函数以确定用户是否最终选择创建

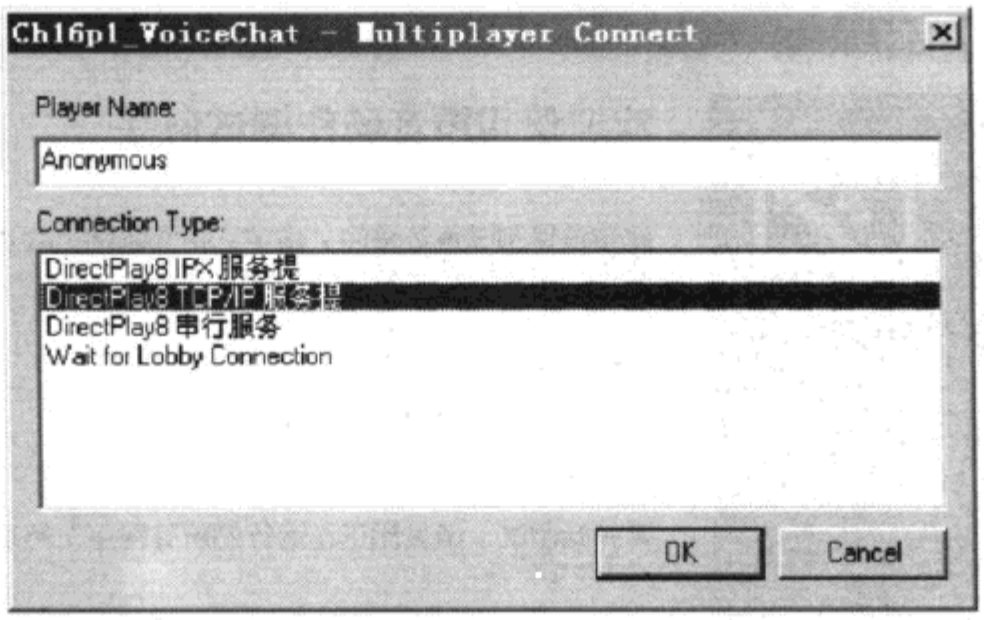


图 16.5 CNetConnectWizard 的连接对话框

一个新的游戏。

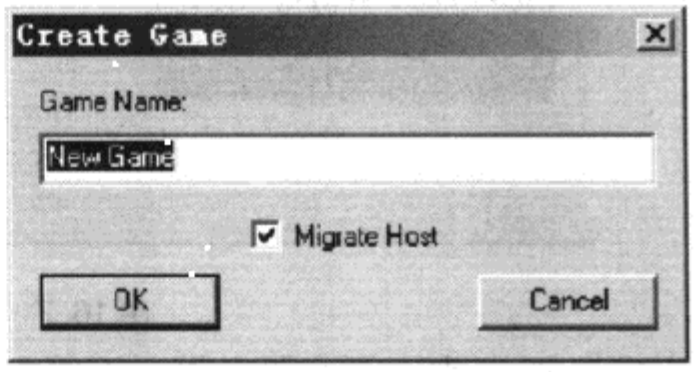


图 16.6 CNetConnectWizard 的新游戏对话框

16.3.4 关闭向导

完成向导后,先调用 Shutdown 进行清理,然后再通过 delete 或者使用 SAFE_DELETE 宏删除对象本身。

16.4 DirectPlay 语音编程

前面已经完成了对 DirectPlay 的概述,现在该进入本章核心部分:使用 DirectPlay 语音实现游戏中的实时语音会话。

从许多方面看来,DirectPlay 语音都可以看作是 DirectPlay 的一个镜像。DirectPlay 语音也有一个消息处理的回调函数,以及分离的语音客户(IDirectPlayVoiceClient)和服务器(IDirectPlayVoiceServer)接口。同时,还有一个名为 IDirectPlayVoiceTest 的接口,它仅有一个函数 CheckAudioSetup。这个函数运行 DirectPlay 语音的音频设置向导,让用户调整播放和录制音量,并指定由哪个设备输入语音(麦克风或线路输入),如图 16.7 所示。

本小节将详细讲述 DirectPlay 语音编程。

16.4.1 初始化 DirectPlay 语音

DirectPlay 语音没有点到点接口,只有 IDirectPlayVoiceServer 及 IDirectPlayVoiceClient。这意味着在初始化 DirectPlay 语音之前,必须先弄清要设定的是语音服务器还是语音客户。没有规定说 DirectPlay 服务器必须同时是 DirectPlay 语音服务器,也没有规定禁止 DirectPlay

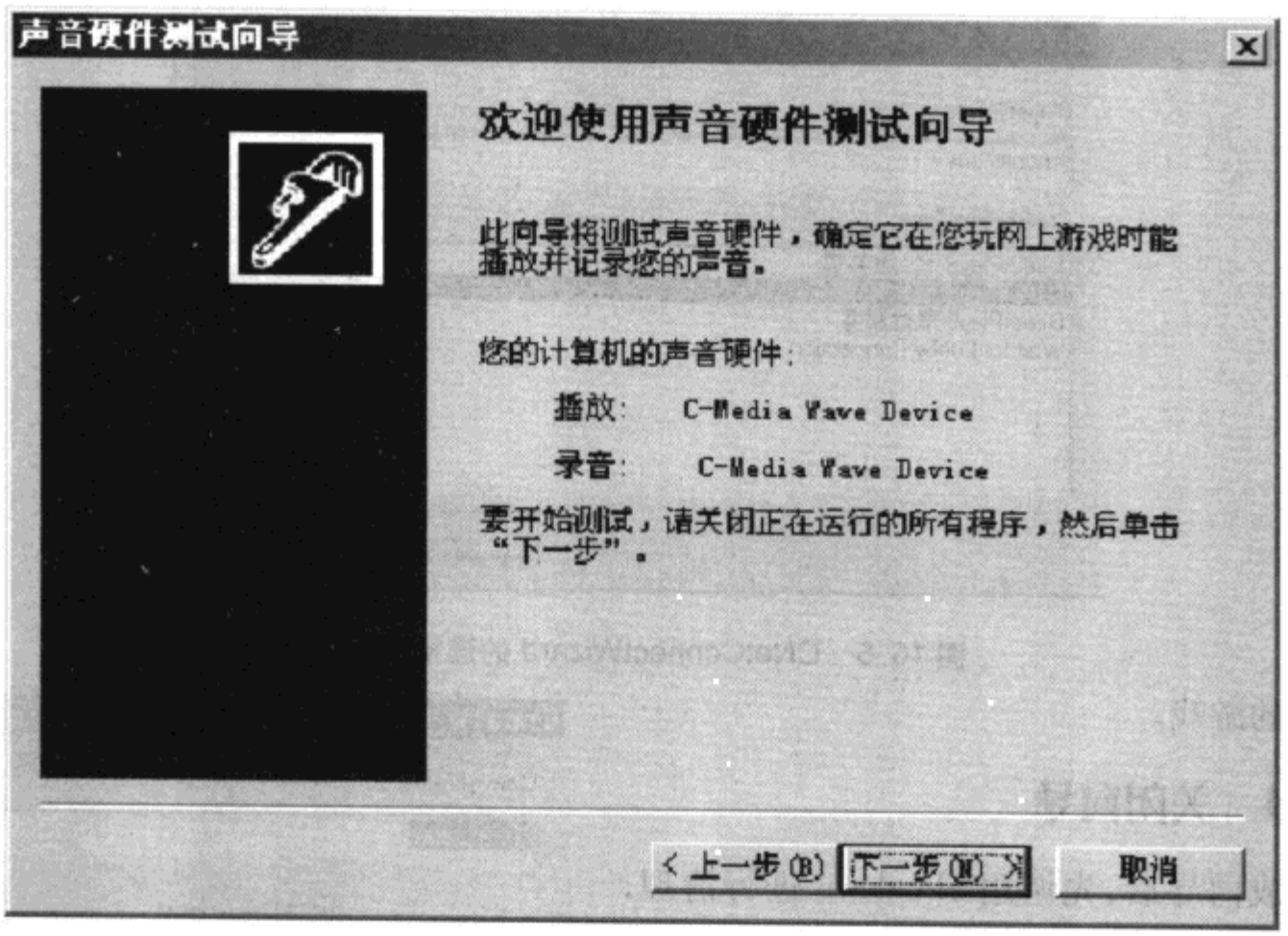


图 16.7 音频硬件测试向导

客户成为 DirectPlay 语音服务器。不管怎样，最简单的判定办法是考虑 DirectPlay 中的角色，如果存在点到点会话，或者是一个 DirectPlay 服务器，则设定计算机为语音服务器；否则，设定其为语音客户。

16.4.2 语音消息

当创建 DirectPlay 语音接口时，也会绑定一个回调函数，以接收 DirectPlay 语音消息。表 16.3 归纳总结了最为常用的语音消息。

16.4.3 设置音频设备

为了获得更好的实时语音交谈效果，麦克风及扬声器的正确设置特别重要。通过创建一个 IDirectPlayVoiceTest 接口并调用其惟一的 CheckAudioSetup 方法，就可以在任意时间打开设置对话框。

也可以在 CheckAudioSetup 的参数中指定用哪个设备播放，哪个设备录音。

表 16.3 常用 DirectPlay 语音消息

DirectPlay 语音消息 ID	描 述
DVMSGID_CREATEVOICEPLAYER	一个已存在的 DirectPlay 玩家激活 DirectPlay 语音并成为 DirectPlay 语音玩家。消息内包含刚成为语音玩家的玩家 ID 号。需要根据这个 ID 去设置附加的玩家参数,以使用 DirectPlay 语音
DVMSGID_DELETEVOICEPLAYER	一个已存在的 DirectPlay 玩家停止使用 DirectPlay 语音。注意,这个消息并不意味着玩家退出游戏,他可能还是一个一般的 DirectPlay 玩家
DVMSGID_PLAYERVOICESTART	在开始播放一个语音流时,DirectPlay 语音将发送此消息
DVMSGID_PLAYERVOICESTOP	在停止播放一个语音流时,DirectPlay 语音将发送此消息
DVMSGID_RECORDSTART	在开始录制本地玩家的语音流时,DirectPlay 语音发送此消息
DVMSGID_RECORDSTOP	在停止录制本地玩家的语音流时,DirectPlay 语音发送此消息
DVMSGID_LOSTFOCUS	当本地玩家失去录制音频的能力时,DirectPlay 语音发送此消息。这通常意味着玩家切换到了另一个应用程序或者本应用程序失去了对音频采集设备的自由控制
DVMSGID_GAINFOCUS	当本地玩家重新获得发送音频的能力时,DirectPlay 语音发送此消息。这通常意味着玩家重新切换回了游戏任务并重新获得了对音频采集设备的控制

16.4.4 半双工及全双工的声卡

语音采集硬件可分为两类:全双工和半双工。全双工意味着声卡可以同时录制及播放声音。半双工即是说录音时不能播放,播放时不能录音。

DirectPlay 语音不支持从半双工的声卡录制声音。这样,只有半双工声卡的游戏玩家只能听到别人的发言,自己不能发言。

现在市场上的绝大部分声卡都是全双工的,所以这不是什么问题。通过调用 DirectPlay

语音客户机或者服务器接口的 GetSoundDeviceConfig 函数就可以确定玩家是运行在一个全双工的还是半双工的声卡之上。这个函数返回一个结构,此结构中的信息全是关于硬件的,其中一项指明设备是否是全双工的。

16.4.5 使用 CnetVoice 类

与 CnetConnectWizard 类一样,CNetVoice 也是由微软发布的辅助类,它包含在 DirectX SDK 的示例中。它被设计用于隐藏处理 DirecPlay 语音的细节,效果很好。

要使用 CNetVoice,只需创建一个实例,设置一些选项,然后调用其 Init 函数即可。下述代码显示了这一过程:

```
g_pNetVoice = new CNetVoice( OnVoiceClientMessage, OnVoiceServerMessage );
// 设置默认的 DirectPlayVoice 设置选项
DVCLIENTCONFIG config;
ZeroMemory( &config, sizeof(config) );
config. dwSize = sizeof(config);
config. dwFlags = DVCLIENTCONFIG_AUTOVOICEACTIVATED |
    DVCLIENTCONFIG_AUTORECORDVOLUME;
config. lPlaybackVolume = DVPLAYBACKVOLUME_DEFAULT;
config. dwBufferQuality = DVBUFFERQUALITY_DEFAULT;
config. dwBufferAggressiveness = DVBUFFERAGGRESSIVENESS_DEFAULT;
config. dwThreshold = DVTHRESHOLD_UNUSED;
config. lRecordVolume = DVRECORDVOLUME_LAST;
config. dwNotifyPeriod = 0;
GUID codecguid = DPVCTGUID_DEFAULT;
// 初始化语音类
g_pNetVoice -> Init( hDlg, g_pNetConnectWizard -> IsHostPlayer( ),
    TRUE, g_pDPPeer, DVSESSIONTYPE_PEER, &codecguid, &config );
```

正如所看到的一样,代码将两个回调函数的名称作为参数提供给 CNetVoice 的构造函数。同时,请注意 DVCLIENTCONFIG 结构的大部分成员都可以简单地设为其默认值。

这些工作完成后,剩下的就是充实客户机和服务器的消息处理函数了。语音结束以后,要记着删除 CNetVoice 对象。

16.4.6 玩家对象的引用计数

在 Ch16p1_VoiceChat 示例程序中,可以看到玩家的语音设置和一般的 DirectPlay 设置

都集成在一个对象里:CPlayer,这是标准做法。大多数游戏都倾向于用一个单独的类或结构来表达一个玩家,这个类或结构包括核心的 DirectPlay 属性,比如玩家姓名,也含括 DirectPlay 语音属性(比如全双工/半双工模式)。

需要注意,当玩家还在进行游戏,而只是暂时停止使用 DirectPlay 语音时,不要删除玩家结构。

Ch16p1_VoiceChat 示例程序通过引用计数来避免错误的发生。当代码接收到一个新玩家消息,它会创建一个玩家对象并设置其引用计数为 1。当代码接收到 DirectPlay 语音的"创建一个语音玩家"消息时,它会寻找对应的玩家结构(保存于从 DPNID 到 CPlayer 对象的 STL 映射中),并增加其引用计数。这样,当所有的东西都创建完成之后,每一个玩家对象的引用计数都为 2,分别记录核心 DirectPlay 和 DirectPlay 语音连接。

在需要删除对象时,代码会递减玩家对象的引用计数。只有当引用计数减为 0 时,代码才真正删除对象并将其从映射中移除。

在这种方式下,先收到哪一个"删除玩家"消息是无关紧要的。只有当 DirectPlay 和 DirectPlay 语音都收到了删除消息以后,玩家对象才会真正被删除。

16.5 作为3D音效缓冲区的语音

DirectPlay 语音有一个很酷的特性,即它允许把语音流绑定到 3D 音效缓冲区上。通过这种方式,如果一个玩家站在你的左边说话,那么听起来,语音就来自于左边。

可以通过调用 DirectPlay 语音客户机接口(IDirectPlayVoiceClient)的 Create3DsoundBuffer 函数进行设置。这个函数将一个 3D 声源绑定到特定的玩家 ID。这样,只需要在缓冲区移动玩家所在的位置,DirectX Audio 就自动完成相应的工作。

提示

注意,在混音服务器拓扑下,DirectPlay 语音与3D 声源的集成是不可能的。因为在混音服务器拓扑下,语音数据在到达你的计算机之前已经过混音,而混音是由服务器完成的,服务器并不知道玩家所在的位置。

16.6 本章小结

本章简单介绍了 DirectPlay 及 DirectPlay 语音。DirectPlay 还有很多其他特性,可以让游戏效果更棒。建议读者自行了解这些功能,并自行编写一些 DirectPlay 应用去体验它们。

PDG

第 17 章

音频可视化

可视化与一本《游戏音频编程》的书有什么关系呢？音频可视化（不论是简单波形、频谱分析、音频修饰，还是 MP3 播放器里流行的音乐驱动特效）通常都与游戏或游戏编程无关。

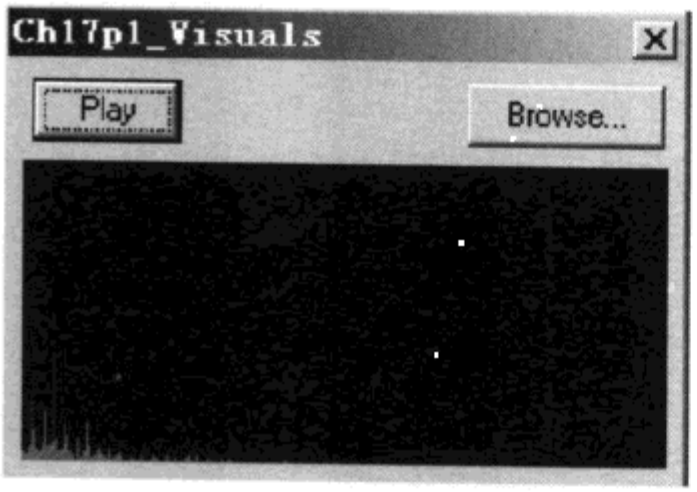


图 17.1　Ch17p1_Visuals 示例程序的截图

然而，音频可视化与游戏音频的许多概念密切相关。音频可视化中的一些核心技术也是高级音频处理的核心技术，包括音频压缩、音频过滤等。对音频可视化，特别是离散傅立叶变换（discrete fourier transform）的正确理解，是成为一个游戏音频编程专家的必备条件。

本章将学习如何设计一个频谱分析器。事实上，一个频谱分析就是一个条形图，它将显示给定声音的频率以及各个频率的音量是多少。图 17.1 是本章示例程序（频谱分析器）的一个截图，读者肯定已经在 MP3 播放器里看到过相似的“频率条形图”。

17.1　离散傅立叶变换

> **提示**
>
> 用数学术语描述，创建一个频谱分析器的真正工作就是将样本数据从时间域转换到频率域。换句话说，你在改变图形的 x 轴。对于样本声音而言，x 轴一般表示时间，但现在要改变为表示频率。

频谱分析的核心思想是用数字序列来表达样本声音。更准确地讲，是用数字序列来表示声波在不同时间点的振幅。现在来构建另一个数字集合，以表达原始样本在不同频率上的振幅。当获得期望的数字集合后，就可以以你喜欢的任何方式去显示它们。可以描绘一个条形图，每一个条表示一个不同的频率，而条的高度代表振幅（大多数 MP3 播放器就是这么做的）；也可以绘制一个 2D 的图形，频率为纵坐标，时间为横坐标，然后使用颜色来区分不同的振幅。重复一遍：在获取频率-振幅数组之后，图形表达形式是可以自由选

择的,这是 GUI 编程器的工作。

获取频率-振幅的一个方法是进行离散傅立叶变换,简称为 DFT。本节内容主要阐述 DFT 的基本理论。

17.1.1 共振

先举一个例子来说明共振的概念。歌剧演唱者在其共振频率上演唱,可以将玻璃酒杯震碎。在这种情况下,演唱者的声音所造成的空气震动使得玻璃酒杯的震动越来越强烈,直到超出玻璃酒杯的承受范围,使其破碎。

也许你无法理解力量是如何积聚起来弄碎玻璃杯的。想象一下,天花板上垂吊一个巨大的球体,球体通过一根结实的绳子吊在天花板上。不论球有多重,只需在合适的时机连续给予其哪怕非常微小的推力,球就会最终摆动起来,而且越摆越高。

同样的道理,如果被推动物体不是球体,而是玻璃杯中的分子;推动物也不是人类的双手,而是通过空气传输的压缩空气波(即声波);如果声波每一个震动的到达时间恰好是可以让分子从其正常位置振动更加强烈的时间,那么慢慢地,声波影响玻璃杯中的大量分子,并使其沿不同方向“摇摆”,直到玻璃杯破碎。

17.1.2 怎样听

共振的属性使得你可以听到声音。人的耳朵就如同一个生物频谱分析仪。耳朵中有一个器官称为耳蜗,呈贝壳状,在耳蜗内有一个膜,称之为基膜(basilar membrane)。基膜的底部窄而硬,只对高频声波产生共振;基膜的另一端宽而柔软,可以对低频声波产生共振;在中间,基膜的不同部分具有不同的共振频率。整个基膜上覆盖着神经末梢,不同频率的声音将引起基膜不同部分的共振,而共振将刺激神经末梢并向大脑发送消息,这样你便“听到了”这个频率。

17.1.3 共振的数学解释

正弦波的和始终为0,换句话说,声波花在0上和0下的时间是相等的。这意味着,如果按照固定的时间间隔对正弦波的振幅进行取样,然后把它们加在一起,最终结果总是接近于0,正负样本最终相互抵消。

有趣的事情发生在按照声波的频率对声波采样时,不管从何时开始采样,最终都会得到一个非0的数字。如果恰巧从声波大于0时开始采样,则下次采样时声波将处于完全相同的位置,这样导致最终的和越来越大。相同地,如果恰巧从声波小于0时采样,最终的和将会越来越小。这就是共振的数学解释。

这意味着如果想判定一个声音是否包含特定频率的声波,只需要用一个声波的点与另

一个声波的点相乘,然后将所有的乘积加起来。这个过程可用如下代码表示:

```
float sinAmp = 0.0f ; // 此变量将包含最终振幅值
for (int k = 0; k < numsamples; k ++ ){
    float x = 2.0f * M_PI * (float)freq * (float)k / (float)numsamples;
    sinAmp += sample[k] * sin(x);
}
```

这是一段假想的代码,它并没有出现在 Ch17p1_Visuals 示例程序中。假定 sumsamples 是输入声波的样本总数,sample 是一个浮点数数组,取值在 -1 ~1 之间。这段代码遍历整个样本数组,将每个样本与 sin(x)相乘,最后将乘积加总到 sinAmp。在循环结束后,sinAmp 包含了声波(在频率 freq 上)的振幅。

为了计算 sin 函数的输入 x,代码将示例中的频率 freq 与 2π(圆周率的 2 倍,大约 6.28)相乘。sin 函数期待弧度(radian)输入,弧度是一种特殊的表示角大小的方式,取值从 0 ~ 2π,而不是 0° ~360°。然后,代码把取得的频率值再乘以 k(循环计数),再除以样本的总数(sumsamples)。代码所表示的是一个在正弦波上迭代的过程。当 k 为 0 时,x 也为 0。当 k 等于 numsamples,k 除以 numsamples 结果为 1,最后 x 为频率的 2π 倍。在这种机制下,会得到频率 freq 的假想波(imaginary wave)上的点,然后再将这个点与声波上的点相乘。当把源声波上的点与对应的频率假想波上的点相乘并将结果相加后,便得到了源声波所包含的 freq 的振幅。实际上,这已经从数学上模拟了共振过程。

17.1.4 单独使用正弦波是不够的

当把输入波与给定频率的正弦波“共振”时,最后取得的振幅跟波形的“相位”有关。相位指声波在图形上左移或者右移的距离。

> **提示**
>
> 事实上,数学上可以证明,任一单一频率,单一振幅的波,不论相位如何,都可以用正弦波和余弦波的和来表示。限于篇幅,这里无法进行严格的数学证明,但我建议访问本书 CD 里提供的几个链接以获取更多信息。

想象有两段波,它们都是 440 Hz 的正弦波,振幅为 5。声波 1 的相位为 0,声波 2 的相位为 π/2 弧度。如果“共振”这两个声波,最后得到的振幅将是不一样的。声波 1 的振幅会比声波 2 的高。当声波 1 位于 5 或者 -5 的波峰时,对应的共振波将会处于 1 或者 -1 的波峰。然而,当声波 2 处于波峰时,对应的共振波为 0。换句话说,共振波“错过”了声波 2 的峰,原因在于声波 2 的位置移动(或者,用更技术化的术语描述,有不同的相位)。当输入波处于最大值时,共振波为 0;当共振波处于最大值时,输入波为 0。这将导致最后计算出的振幅

小于实际值。

要解决这个问题,需要使用余弦波。余弦波的波形与正弦完全相同,区别是余弦波从1开始,而正弦波从0开始(如图17.2所示)。换句话说,余弦波是改变了相位的正弦波。

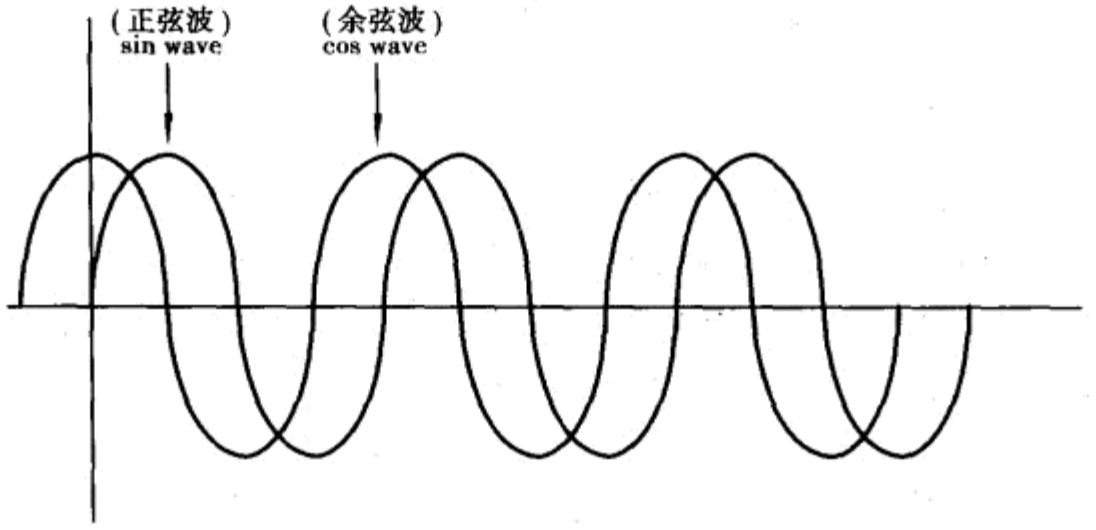


图17.2 余弦波是改变了相位的正弦波

如果同时将输入波与正弦波和余弦波共振,而不论输入波的相位(位移的量)如何,都会得到相同的振幅。这是因为正弦波和余弦波相互交叠,当一个为0时,另一个就为最大值,这样就永远不会错过输入波的波峰。

同时使用正弦波和余弦波的代码,如下所示:

```
float sinamp = 0.0f;
float cosamp = 0.0f;
for (int k = 0; k < sumsamples; k ++) {
  float x = 2.0f * M_PI * (float)freq * (float)k / (float)numsamples;
  sinamp -= sample[k] * sin(x);
  cosamp += sample[k] * cos(x);
}
float amp = sqrt( sinamp * sinamp + cosamp * cosamp );
```

这段代码与在前几节中看到的代码相似,区别是以前的代码仅使用正弦波,而这段代码同时使用正弦波和余弦波(对sinamp执行减操作以使得正弦波从0下开始,就如同余弦波一样)。同时,还有新的代码行计算sinamp的平方与cosamp的平方之和的平方根。

事实上,这是一种隐含的距离计算。假设需要计算一个特定点,比如(3,4)到原点的距离,则应将两个坐标值取平方,然后相加,再求平方根。3的平方是9,4的平方是16,9加上16等于25,这样,最终的距离是25的平方根,即5。如果在一个2D图形里以sinamp作为横坐标,以cosamp作为纵坐标绘制一条曲线,就可以计算振幅,即坐标点(sinamp,cosamp)到原点的距离。如果还是不太明白,请自己补充一些数学背景知识,建议浏览光盘中提供的

几个链接。

讲了这么多，读者应已经具备实施一个频谱分析器的最重要的知识：即如何判定在一个 WAV 文件中一个频率有“多少”。懂得这一点之后，需要做的就是对不同的频率值计算振幅，最终得到一个适合于图像显示的输出数组。

17.1.5 Nyquist 数

“我应该使用哪些频率?”，回答这个问题，必须首先了解一个特殊的数 Nyquist。Nyquist 数是指在一定的采样频率下可以获取的最高频率。

例如，假设一个声源的频率是 10 Hz，用 20 Hz 的频率进行采样。这里，为说明概念，故意将数字取得较小，实际上，大多数声音的采样频率都是 22 050 Hz 或者 44 100 Hz。在本例中，我们不管质量，以 20 Hz 进行采样。

用 20 Hz 的采样频率对 10 Hz 声波进行采样不会得到高质量的样本，但仍可以采集到 10 Hz 的原始频率。现在，假设以 20 Hz 的采样频率对 30 Hz 的声波采样。最后获得的样本将非常奇怪，因为声波起伏得过快，无法看见他们的变化。这称之为混淆(aliasing)现象，这与快速转动的直升机螺旋桨或汽车轮胎有时看起来是在反转是一个道理。实际上，它们并没有反转，只是人的视觉采样频率过低以至于无法正确感知其运动，所以它们看起来是在反转。

Nyquist 数是在给定的采样频率下，在不会产生混淆现象的前提下，可以采样的最高频率。它刚好等于采样频率的一半。即，对于 44 100 Hz 的采样频率而言，其最高可采频率为22 050 Hz。

17.1.6 离散傅立叶变换

前面已如道如何判定一个声波包含了指定的频率，理解了 Nyquist 数，知道需要计算的最高频率。现在，所需要做的就是把已经熟悉的代码包装在另一个 for 循环里：

```
for (int bin =0; bin <= numsamples/2; bin ++ ) {
    float cosAmp =0.0f;
    float sinAmp =0.0f;
    for ( int k =0; k < numsamples; k ++ ) {
      float x =2.0f * M_PI * (float)bin * (float)k / (float)numsamples;
      sinAmp += m_Input[k] * sin(x);
      cosAmp += m_Input[k] * cos(x);
    }
    m_AmpOutput[bin] =sqrt( sinAmp * sinAmp + cosAmp * cosAmp );
```

}

在这里,freq 变量变成了 bin,程序从 0 循环到 Nyquist,而最终的振幅变为频率的振幅数组。

17.1.7 频谱分析器动画

上一节代码会产生一个包含足够多频率的输入声波。要实现一个动态的频谱分析器,只需要计算从当前样本声波位置开始的一小段(最好为 512 字节)输入声波的频率。在后续几节里,将了解更详细的过程。

17.2 示例程序

读者们已经了解了关于频谱分析器的所有背景理论知识,现在可以去实现它了。后续小节将分析 Ch17p1_Visuals 示例程序中的代码。

图 17.1 是 Ch17p1_Visuals 示例程序的一个截图。如图所示,它的 GUI 非常简洁,只有两个按钮,分别是 Browse 和 Play,按钮上方有一个用于显示频谱分析结果的矩形。

17.2.1 装载 WAVE

同第 2 章一样,本示例程序使用底层 DirectX 音频接口 DirectSound。之所以选择 DirectSound 而不是 DirectMusic,主要是因为使用 DirectSound 更容易确定任一时刻正从扬声器中播出是样本的哪一个字节。这样,对于本书而言,这也是一个对称的结尾,我们从低层 API 讲起,现在又回到低层 API。

不幸的是,使用 DirectSound 必须自行装载 WAV 文件,因此无法将其简单地装载到一个 DirectMusic 段中。简单起见,可以使用 CWAVFile 中的代码去装载文件,如下所示:

```
CWAVFile wavfile;
wavfile. Load( filename) ;
CreateSecondaryBuffer( &g_pWAVBuffer, wavfile) ;
unsigned char * ptr1 = NULL;
unsigned long len1 =0;
g_pWAVBuffer -> Lock(0, 0, (void ** )&ptr1, &len1, NULL, NULL,
    DSBLOCK_ENTIREBUFFER) ;
memcpy( ptr1, wavfile. GetData( ), wavfile. m_DataLen) ;
g_pWAVBuffer -> Unlock( ptr1, len1, NULL, NULL) ;
```

这段代码是从 Ch17p1_Visuals 示例程序的 LoadSound 函数中摘录出来的。这里可以看

到代码将 WAV 装到 wavfile 变量，然后锁定一个 DirectSound 辅缓冲区，再将内容填充进去，并最终将其解锁（如果希望重温如何将一个 DirectSound 缓冲区锁定和解锁，请参看第 2 章）。

17.2.2 CdiscreteFourierTransform 类

Ch17p1_Visuals 依赖于一个新类 CDiscreteFourierTransform。这个新类包含了用于进行离散傅立叶变换的所有数据和方法。下面是类的声明：

```
class CDiscreteFourierTransform
{
public:
   CDiscreteFourierTransform(int bufsize, int samplerate, int bitspersample, int channels);
   virtual ~CDiscreteFourierTransform();
   void LoadInputData(unsigned char * data);
   void PerformSlowFourierTransform();
   float GetAmp(int ndx)   { return(ndx > m_BufSize ? 0.0 : m_AmpOutput[ndx]); }
   float GetFreq(int ndx) { return(ndx > m_BufSize ? 0.0 : m_FreqOutput[ndx]); }
   int m_SampleRate;
   int m_NumChannels;
   int m_BitsPerSample;
protected:
   int m_BufSize;
   float * m_Input;
   float * m_FreqOutput;
   float * m_AmpOutput;
};
```

这个类的主要成员是 3 个浮点指针，m_Input、m_FreqOutput 及 m_AmpOutput。这些指针所指向的是动态分配的浮点数组，数组的大小取决于缓冲区的大小（m_BufSize）。m_Input 数组用于存储输入样本；m_AmpOutput 则用于存储不同频率的振幅，这是频谱分析器的核心；m_FreqOut 用于告知客户端代码指定下标的频率是多少。如果要给频谱分析器添加标签（labels），这将很有用。

（1）构建数组

浮点数组在对象被构造时创建，如下列代码所示：

```
CDiscreteFourierTransform::CDiscreteFourierTransform(
  int bufsize, int samplerate, int bitspersample, int channels):
  m_BufSize(bufsize), m_SampleRate(samplerate),
  m_BitsPerSample(bitspersample), m_NumChannels(channels)
{
  // 为缓冲区分配空间
  m_Input = new float[bufsize];
  m_FreqOutput = new float[(bufsize/2) + 1];
  m_AmpOutput = new float[(bufsize/2) + 1];
  // 我们已经知道每个 bin 所对应的频率
  for (int bin = 0; bin < m_BufSize/2; bin ++) {
    m_FreqOutput[bin] = (float)bin * m_SampleRate / m_BufSize;
  }
}
```

注意,代码借这个机会创建频率输出数据 m_FreqOutput。由于这个数组是不发生改变的,所以此时是计算其值的最佳时机。

(2)装载值集

在客户端代码试图更新频谱分析器时,它会向 CDiscreteFourierTransform 装载一个新的数值集合。这个数值集是声波的一个片断,开始于当前播放位置并延续 BUFFERSIZE(值为 512 的常量整数)个样本。在开始 DFT 计算之前,CDiscreteFourierTransform 必须先将样本数组从 0 ~ 255 整数(对于 16 比特样本,为 0 ~ 65 535)转换为 -1.0 ~ 1.0 的浮点数。此过程如下列代码所示:

```
void CDiscreteFourierTransform::LoadInputData(unsigned char * data)
{
  // 将 0 ~ 255 的整数转换为 -1..1 的浮点数
  if (m_BitsPerSample == 8) {
    for (int q = 0; q < m_BufSize; q ++) {
      float d = data[q * m_NumChannels];
      m_Input[q] = (d - 127.0f)/127.0f;
      if (m_Input[q] <= -1.0f) m_Input[q] = -1.0f;
      if (m_Input[q] > 1.0f) m_Input[q] = 1.0f;
    }
```

```
    }
    else if (m_BitsPerSample == 16) {
      short int * sidata = reinterpret_cast < short int * > (data);
      for (int q = 0; q < m_BufSize; q ++ ) {
        float d = sidata[ q * m_NumChannels ];
        m_Input[ q ] = (d - 32767.0f)/32767.0f;
        if (m_Input[ q ] < -1.0f) m_Input[ q ] = -1.0f;
        if (m_Input[ q ] >1.0f) m_Input[ q ] =1.0f;
      }
    }
    else{
      // 不支持
      for(int q = 0; q < m_BufSize; q ++ ) { m_Input[ q ] = 0.0f; }
    }
}
```

可以看到,这里使用了一些指针技巧。根据 m_BitsPerSample 值的不同,将输入的 data 变量当作 8 比特字节数组或者 16 比特短整型数组。同时,要注意到 m_NumChannels 变量的使用,对于立体声样本,这里仅使用第一个声道。

(3)DFT 计算

数组创建好了,m_Input 数组也填满了浮点值,此时客户端代码将调用 PerformSlowFourierTransform进行 DFT 计算,并填充 m_AmpOutput 数组:

```
void CDiscreteFourierTransform :: PerformSlowFourierTransform( )
{
  for (int bin = 0; bin <= m_BufSize/2; bin ++ ) {
    float cosAmp = 0.0f;
    float sinAmp = 0.0f;
    for (int k = 0; k < m_BufSize; k ++ ) {
      float x = 2.0f * M_PI * (float)bin * (float)k / (float)m_BufSize;
      sinAmp += m_Input[ k ] * sin(x);
      cosAmp += m_Input[ k ] * cos(x);
    }
    m_AmpOutput[ bin ] = sqrt( sinAmp * sinAmp + cosAmp * cosAmp);
```

```
    }
}
```

此时,读者对这段代码应当很熟悉了。它基于本章开始部分所介绍的理论。

17.3 图形渲染

最后,介绍使用 CDiscreteFourierTransform 的 m_AmpOutput 数组对频谱分析器进行渲染的代码:

```
void RenderSpectrum(HDC hdc, HWND hDlg)
{
   HWND framewin = GetDlgItem(hDlg, IDC_FRAME);
   RECT framerect;
   int framewidth,frameheight;
   GetWindowRect(framewin, &framerect);
   ScreenToClient(hDlg, (POINT *)&framerect.left);
   ScreenToClient(hDlg, (POINT *)&framerect.right);
   framewidth = framerect.right - framerect.left;
   frameheight = framerect.bottom - framerect.top;
   // 绘制每一个条形
   int n = 0;
   for (float x = framerect.left; x < framerect.right;
        x += framewidth / ((float)BUFFERSIZE/2.0f), n ++) {
      RECT barrect;
      barrect.left = x;
      barrect.right = x + (framewidth / ((float)BUFFERSIZE/2.0f));
      barrect.bottom = framerect.bottom;
      barrect.top = framerect.bottom - (g_FT -> GetAmp(n) / 50.0f * (float)
frameheight);
      if (barrect.top < framerect.top) { barrect.top = framerect.top; }
      FillRect(hdc, &barrect, (HBRUSH)GetStockObject(GRAY_BRUSH));
      // 将上半部分填黑
      if (barrect.top > framerect.top) {
         barrect.bottom = barrect.top;
```

```
            barrect.top = framerect.top;
            FillRect(hdc, &barrect, (HBRUSH)GetStockObject(BLACK_BRUSH));
        }
    }
}
```

这是一段基于 Win32 API GDI 的代码,稍稍有点难理解。它从输出矩形的左端循环到右端,计算每个条形的高度并绘制两个矩形,一个灰色(条形的底部),一个黑色(条形的顶部)。使用这种方式,图像闪烁被降低了,因为我们没有全部清除整个矩形并在其之上重新绘制。

如果还想更专业一点,建议读者做更多的工作。常见的一个效果是给条形添加一个红色到绿色的坡度变化,使得频谱分析器更亮丽些。

17.4 快速傅立叶变换

> **提示**
>
> 如果运行流行的 SETI@ Home 屏保程序,读者可能会发现它用了大量的时间去计算"FFT"。这些 FFT 事实上就是快速傅立叶变换——屏保程序使用快速傅立叶变换去降低信号源不同频率的振幅值。

本章所提供的代码有一个问题:太慢了。本书尽量使得代码简单易懂,但这意味着代码没有经过优化,除了用于讲述概念之外,并不适合于实用场合。仅仅是把 BUFFERSIZE 设为 1 024 这样的小值,会拖垮最新的硬件。

要解决 DFT 的问题,必须提速。1965 年 Cooley 和 Turkey 发明了一种可以将 DFT 提速几个数量级的神奇算法,称之为快速傅立叶变换,或简称 FFT。FFT 完成的工作与离散傅立叶变换一样,区别在于,它大大地提高了计算速度。

限于篇幅,这里无法详述 FFT,但在本书附带的光盘中提供了几个链接,它们将讲解 FFT 是怎么回事及如何实现。这里学习的目标就是完全理解它,并在 CDiscreteFourierTransform 中增加一个 FFT 方法,为真正实时的应用作好准备。

17.5 本章小结

希望本章已经向读者传递了实现一个频谱分析器的核心概念。更理想化一点,希望本章让读者更加懂得欣赏声波简洁的优雅。人们所听到的东西:从歌剧、爆炸声到人类的语

音，都是源于不同频率的声波引起了耳蜗的不同部分的共振。

关于离散傅立叶变换，还有许多内容没有描述。事实上，DFT 可以从很多不同的视角进行解释，其中有些解释需要用到很高深的数学知识。如果你是那种喜欢在布满数字和积分公式的道路上散步的人，建议你了解关于 DFT 和 FFT 的更多数学细节。

结束语

到这里就是结尾。希望读者们已经牢牢掌握了关于游戏音频编程的基础知识，并找到了将所学到的音频知识应用于自己开发游戏的动力。

随着本书学习进度而逐步建立起来的音频引擎等待着读者将它应用到具体游戏的声效编程中去。在本书配套光盘的“FinalAudioEngine”中可以获得最终的音频引擎，也可以从每一章的介绍中去理解这个音频引擎。如果仅希望音频引擎具备某部分功能，也可以只取到某一章为止的特性。

建议读者去浏览 CD 中提供的链接。这些链接是最好的资源，它们可以帮助提升对游戏音频的理解，并为音频引擎添加更多优异的特性。这个引擎现在是属于你的了，请拿去、修改并完善它，并将其应用于你自己的游戏项目中。

祝工作愉快！



PDG

附录 A

创建动态音乐

现在,请从程序员的角色转变到音乐家的角色。本附录将介绍完成创建简单乐谱的整个过程。

本附录重点解释如何创建动态音乐,而不关心 DirectMusic Producer 的工作机理。偶尔给出一张插图以方便读者理解。如果有关于 DirectMusic Producer GUI 的问题,可以寻求在线帮助或者阅读微软提供的介绍。

现实生活中的音乐会比附录中所创建的音乐更复杂而有趣。本附录重点介绍创建音乐乐谱的过程和方法,对所创建的音乐乐谱将尽可能简化。到本附录结尾,你将仅获得只有两三个乐部的动态音乐片断。当然,读完附录后,你可以随时重新开始并充实你所创建的音乐!

附录将从学习如何安装 DirectMusic Producer,以及如何用它创建静态音乐开始。然后学习如何通过向静态音乐添加元素使其变成动态音乐的技巧。

A.1 安装 DirectMusic Producer

首先,在 DirectX SDK 的 CD 盘上找到 DirectMusic Producer 的安装程序,按引导提示完成安装。安装过程没什么特别的需要介绍,如果遇到麻烦,请查阅 DirectMusic Producer 的帮助文档。

A.1.1 设置 MIDI 键盘

在 DirectMusic Producer 中设置 MIDI 键盘,只需要点击工具栏上一个类似于 MIDI 接口的按钮,就可以改变 MIDI 设置。参见图 A.1。

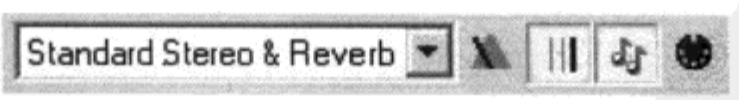


图 A.1 工具栏上类似 MIDI 接口的按钮

提示

如果打算认真地谱一些曲子,但却没有 MIDI 键盘,那么,请赶紧去买一个。试图用鼠标来操作 DirectMusic 将非常困难。至少需要有一个小的八度键盘。

如果没有 MIDI 键盘,在 CD 的 Goodies 目录中有一个程序,可以把 QWERTY 键盘模拟成 MIDI 键盘。

这个对话框中最重要的设置是 MIDI 输入下拉式选项,不要改变其他配置(回声、延迟与合成器)。如果你真想这么做,DirectMusic Producer 中的帮助会告诉你它们是什么。

读者也许希望在敲 MIDI 键盘时能听到演奏音符的声音,但 DirectMusic Producer 不支持这么做。要测试 MIDI 键盘输入,可以参照图 A.2 所示方法,按照如下所述步骤进行。

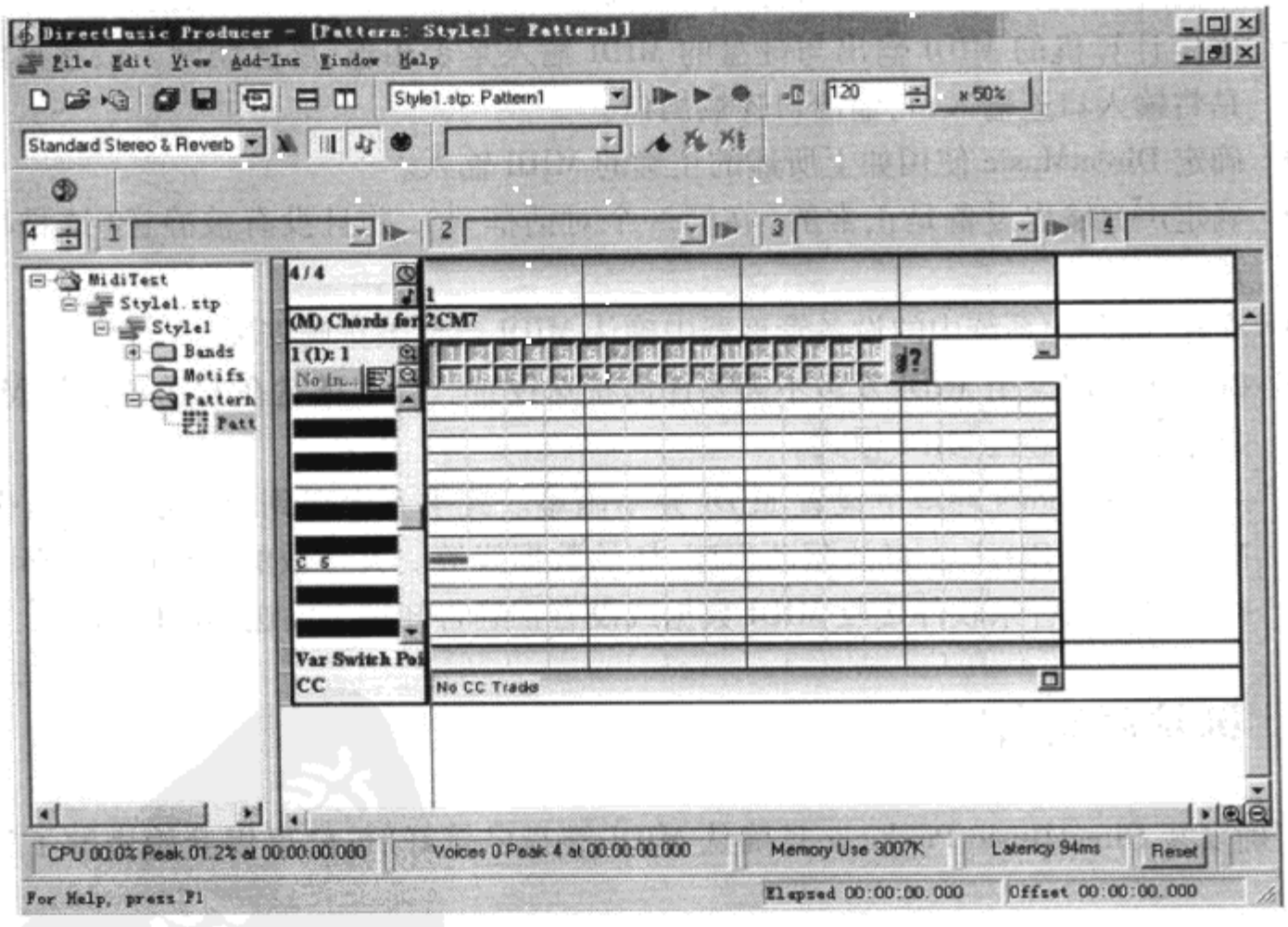


图 A.2 用于测试 MIDI 输入是否有效的测试项目

①点击工具栏上的“新建”按钮并选择风格。DirectMusic 会提示你输入项目的名称,这里,我们根据示例的目的命名为“MidiTest”。

②现在,在屏幕的左端窗口有一个对象树图。打开 Pattern 目录,并双击 Pattern1。

③在屏幕右端有一个标记为“1(1):1”的蓝色方块。双击它后,一个键盘将会出现在左下方。

④单击键盘上的一个键。将会听到相关的音符声,并且整个键区变成亮黄色。你现在应该可以敲击键盘并听到声音了。

⑤在文件菜单中选择关闭项目,不保存。

⑥浏览找到在第 1 步中创建的项目,删除它(如果你没有设定位置,它应该位于 My Documents\DMUSProducer 目录中)。

A.1.2 如 MIDI 键盘不工作

对于 MIDI 新手,且 MIDI 键盘无法与 DirectMusic 沟通,以下是一些帮助:

- 确定计算机的 MIDI 输出与键盘的 MIDI 输入是否相连,反之相同。最常见的问题是将输入口连输入口,输出口连输出口。
- 确定 DirectMusic 使用如上所述的正确的 MIDI 输入。
- 确定声音输出设备是正常的(在同一个对话框里),并且没有被静音(这种事发生过)。
- 在控制面板/系统中的设备管理器中确认 MIDI 端口被正确安装。
- 确定所用键盘用 MIDI 发声不需要任何特殊设置。有些键盘在发送数据至 MIDI 端口之前需要进入 MIDI 模式。
- 一般在 Windows 环境下设置 MIDI 并不困难。几乎不会遇到由底层问题所引起的 MIDI 错误:IRQ、bug 多的驱动程序、坏的数据线等。大多数的 MIDI 问题都在于没有正确告知音乐软件接收 MIDI 数据或没有正确告知键盘发送 MIDI 数据。

A.2 创建静态音乐

正确安装 DirectMusic Producer 且确认 MIDI 键盘已接好后,就可以开始谱歌了! 过程如下:

①选择将在音乐中使用的乐器。

②创建一个乐队(即设置乐器的默认音量及排位)。

③创建音乐的模式(通常是工作量最大的一步)。

④用段将模式串起来。

⑤如果需要,做些调整。

⑥保存并试听。

本节试图创建一般的音乐,在最简单的层次上走完这些步骤,没有动态特性。

在本附录的后面部分,可以重新编排本节中所创建的静态音乐,使其变成动态音乐。

提示

如果不想一步一步地走完本节的所有步骤,可以阅读本书附带 CD 中所含的项目文件。

A.2.1 创建对象

每段动态音乐都由一个新的 DirectMusic 项目开始。

提示

新建对话框中的对象列表是可以多选的,所以以后你可以在一步中同时选择 Style 和项目以创建一个项目和一个风格。

首先从文件下拉菜单中选择新建“项目”。然后,在同一个对话框中创建一个“风格”。注意,当你创建一个新的风格时,你将同时得到一个新的乐队和模式。

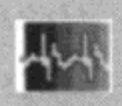
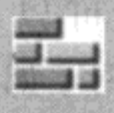
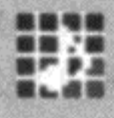
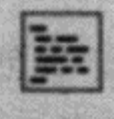
这个例子很小,所以你可以不修改 DirectMusic 给对象的默认名字,如 band1,pattern1等等,以使它们得到更好描述。当然,如果你谱的是真实的音乐,一定要养成给模式一个恰当描述性名字的习惯。

A.2.2 图标、扩展名及文件格式

在进行下一步之前,应该知道如何识别你所创建的对象。为了帮助理解 DirectMusic 树形图中的图标,表 A.1 给出了 DirectMusic Producer 中对象的图标、文件扩展名、以及简短的描述。

把 DirectMusic 对象存到硬盘上有两种不同的方式:常态格式及运行格式。在创建音乐时,用常态存储,但当完成了创建工作,并试图在应用中听到音乐时,应该存为运行格式。运行格式不包含编辑信息,因此其文件通常比常态格式更小。

表 A.1 DirectMusic Producer 中的对象

图标	对象	常态格式	运行格式
	项目,它是最高层次的对象	PRO	N/A
	DLS 集合。存储自定义的乐器和样本(鼓声循环等)	DLP	DLS
	DLS 集合中的乐器	包含在 DLS 中	包含在 DLS 中
	DLS 集合中的 WAV 文件	WAV	包含在 DLS 中
	风格(style)。一种模式,它是乐旨和乐队的集合	STP	STY
	乐队。即在一个模式中使用的乐器	BNP	BND
	段。告知 DirectMusic 播放哪一个风格(以及风格所含的模式)	SGP	SGT
	音频路径。比如,立体声、单声道声、带混响的立体声等	AUP	AUD
	乐旨。由游戏驱动的音乐音效(其调性是被保证的!)	包含在风格中	包含在风格中
	模式(Pattern)。包含长度为 1 小节到 n 小节的一段音乐	包含在风格中	包含在风格中
	脚本。告知 DirectMusic 如何播放音效	SPP	SPT
	WAV 文件。用于存储音效或样本音乐,比如,一段吉它音效	WVP	WAV

（续表）

图标	对象	常态格式	运行格式
	和弦地图（Chordmap）。用于动态音乐合成	CDP	CDM
	容器。它包含自定义对象	COP	CON
	工具图。给执行通道应用工具和效果	TGP	TGR

A.2.3 创建 DLS 集合

如果想用一段鼓声循环来增强某首歌的风格，就需要一个 DLS 集合；如果所需要的乐器在通用 MIDI 声音集合中可以找到，那么就不需要创建 DLS 集合。

点击“新建”按钮并选择 DLS 集合选项，以创建一个 DLS 集合。在树形图中可以看到它分为两个目录：乐器和 WAV。

WAV 目录包含了用于创建乐器的原始 WAV 样本。这意味着要做的第一件事情是把一个 WAV 样本加到 WAV 目录中。右击目录然后选择 Insert Wave，接下来浏览到相应目录并选定希望插入的 WAV 文件（本例中，指定 CD 上的 drumloop. wav）。下一步，双击刚刚创建的 drumloop 图标，则 WAV 文件会在右端的面板中展开。

DirectMusic Producer 提供一些基本的 WAV 编辑功能，对于复杂一些的编辑任务，可以通过诸如 GoldWave 或者 Sound Forge 之类的工具完成。

为了播放声音，首先要确定 WAV 文件被正确加载。在工具栏上接近中间的位置，有几个播放按钮。其中，带竖条的表示从开始处播放，不带竖条的表示从当前位置播放。不必每次都把一个对象打开后再播放，可以在播放按钮的左边的组合框中选择希望播放的对象。

装载了 WAV 文件后，就可以创建乐器了。乐器可以基于 WAV 文件自建，建好后可在树形视图中找到。DirectMusic 遵循 DLS 标准，但是，如何把 WAV 文件转换为乐器，DLS 标准有很多选择。根据标准，不同音调范围可以指定为不同的 WAV 文件。例如，可以用一个 WAV 文件表达乐器的低八度，而用另一个 WAV 文件表达乐器的高八度。也可以为乐器的每一个音符指定一个 WAV 文件。

本例中使用一个鼓声循环，所以并不在意音调，鼓声循环可以在任何时候以正确的频

率播放。通常,应通过 WAV 编辑页中的音调范围编辑框把高低音调范围都设为样本的默认频率 C5(第五个八度的 C 音符),即 DirectMusic 假定采用 C5 音符播放 WAV 文件(除非另外指定)。如果指定一个低一点的播放音符 B4,则 DirectMusic 会把样本的音调降到那个音符;如果指定为 C6,则 DirectMusic 将 WAV 样本提高八度来播放;在默认 C5 时,DirectMusic 不会改变样本的音调。

现在,把注意力转到乐器框中的一系列控件上(见图 A.3)。选中 Drums 复选框以告知 DirectMusic 这个乐器是一个真实的鼓工具箱。鼓工具箱是一种特殊乐器,表示一系列鼓的集合,每一个音符都是一个类型不同的鼓。比如,C5 可能是个低音鼓,下一个音(C#5)可能是一个小军鼓,D5 则可能是个铜钹(hi-hat)。你可以创建许多不同类型的鼓,然后把它们都指定到同一个乐器声中。

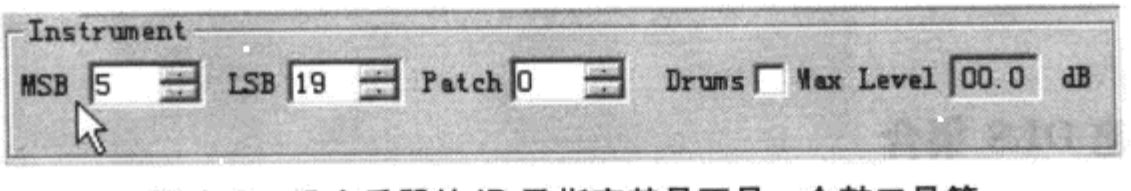


图 A.3 设定乐器的 ID 及指定其是否是一个鼓工具箱

这样使得录入鼓轨迹非常容易。如果每一个鼓都是一个单独的乐器,那你就必须为它们创建不同的乐部。这意味着,你必须先录低音鼓,然后是小军鼓,接着是铜钹,且每次都要单独进行。然而,如果有一个鼓工具箱,你可以通过敲击键盘上的不同按键,一次性地录入所有鼓的轨迹。虽然与敲真鼓仍有不同,但已经比单个录入容易多了。

提示

想了解更多关于 MSB、LSB 及 patch 数字的信息,请查阅 DirectMusic 文档,路径:Playback and Recording \ Patch Numbers and Bank Select。

紧挨着 Drums 复选框左边的 3 个数字选项共同指定乐器的 ID。MSB 意为最高有效字节(Most Significant Byte),LSB 意为最低有效字节(Least Significant Byte)。我的建议是,除非你有超过 127 个的自定义乐器,设定 MSB 及补丁字节(patch)的值为 0,用 LSB 单独表示乐器的 ID。一种可取方法是把相似的乐器分成组。比如,弦乐器从 LSB 10 开始,木管乐器从 LSB 20 开始等。

对于本例而言,乐器 ID 无关紧要,所以可以随意指定为 MSB 0、LSB 1、Patch 0,简写为(0,1,0)。

创建自定义鼓声循环的下一步是确定正确设定了发声方法(articulation)。术语发声方法指一个音符在时间轴上的音量变化。不同的乐器发声方法不同;比如,钢琴音符的发声会快速增强,并慢慢衰减。而由另外一种乐器发出的音符声则可能会随时间缓慢增强。由模拟合成器发出的嘟嘟声则会迅速增强和消失,就如同开关的开合一样,没有衰减期。

创建一种逼真的乐器,需要设置乐器音量的触发期(attack)、衰减期(decay)、保持期(sustain)和释放期(release)。这些都是 MIDI 的术语,触发期指音符以多快的速度达到最高音量,衰减期指以多快的速度从最高音量减弱,保持期指音量恒定的时间,而释放期指音量以多快的速度完全消失(参见图 A.4)。请注意,对于乐器音调,同样有 4 个 ADSR 值,虽然这些值很少改变。

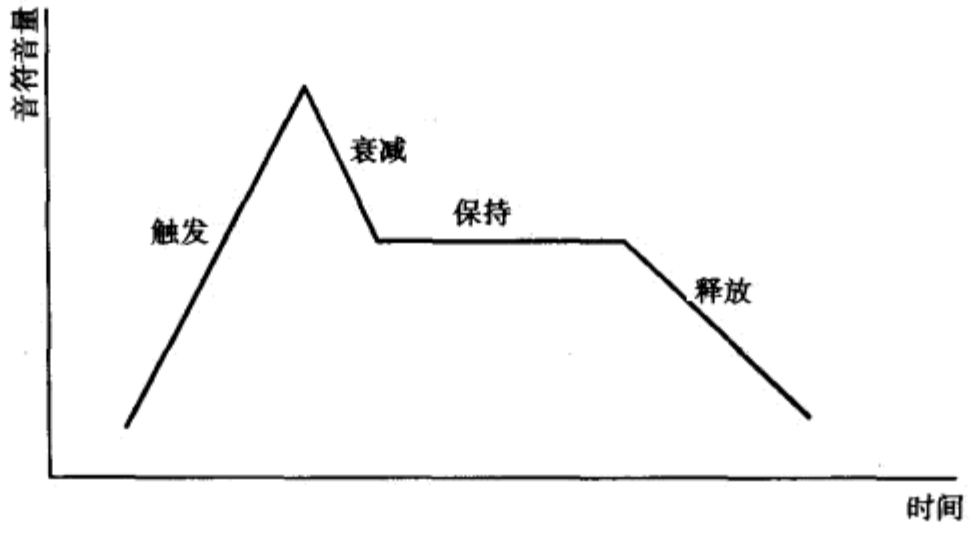


图 A.4 DLS 集合中的每个乐器都有 ADSR(触发、衰减、保持、释放)属性

由于现在所创建的乐曲只是一个鼓声循环,并不希望玩什么技巧,此时设置触发、衰减、释放值为 0.001,保持值为 100%。即告知 DirectMusic 音符发声瞬间增强,然后保持全音量,最后瞬间消失,这很接近真正的鼓声。

创建自定义乐器的最后一步是取消 Note Overlap Allowed(允许音符叠加)选择项,即告知 DirectMusic:如果乐器正在演奏,在开始演奏下一个音符之前必须先停止上一个音符的声音,不允许两个音符叠加。换句话说,这个乐器仅有级别一的复调能力——同一时间仅有一个实例的声音演奏。这样的设置适合于鼓样本和鼓声循环,但不适于其他常见乐器。

提示

在编写本书时,DLS 标准有两个版本:DLS Level 1 和 DLS Level 2。DLS Level 1 标准支持乐器的触发、衰减、保持、释放 4 个参数,而 DLS Level 2 支持更精确的调整,它增加了坚持(hold)、延迟(delay)两个参数(参见图 A.5)。如果需要使用这些额外参数,请取消 DLS 1 选择框。注意,此时你的乐器仅能在支持 DLS 2(DirectMusic,从 DirectX8.0 开始支持)的软件合成器上工作。要了解关于 DLS 1 与 DLS 2 差异的更多信息,请查询 DirectMusic Producer的在线帮助。位置:Playback and Recording\DLS Designer\DLS Level 1 and Level 2。

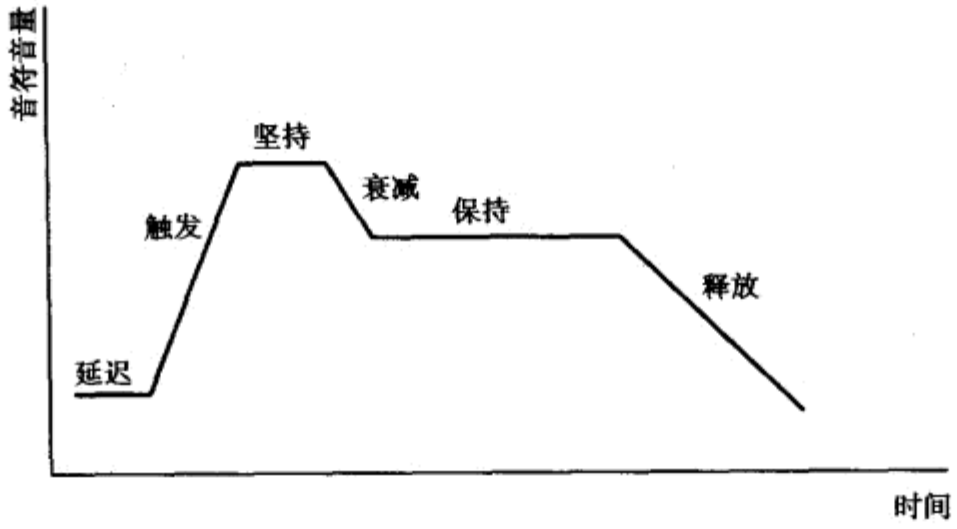


图 A.5 DLS Level 2 乐器具有的属性

敲击 MIDI 键盘上的 C5 键几次,注意观察:当键盘按下时样本音如何开始,当抬起手指时样本音如何结束。如果不熟悉整个 ADSR 逻辑如何工作,可适当修改参数,并测试当敲击 C5 键时所发出的声音发生了怎样的变化。

现在,我们已经为示例歌曲创建了一种自定义的 DLS 乐器。对话框中还有一些选项没有谈及,因为它们很少用到且同本例没有什么关系。如果想了解这些选项,请查询 DirectMusic Producer 帮助文档,路径:Playback and Recording\DLS Designer。

提示

简单假定:只有当按要求播放声效段时,才需要创建自定义 DLS 集合。你可以创建一个 DLS 集合并置于示例中,然后创建一个段,包含一个风格及一个模式,且在仅有的那个模式中仅包含一个音符——希望演奏 C5。

当然,更为简单的方式是创建一个基于 WAV 的段(段的内容为一个 WAV 文件,而不是音符)。DirectMusic 支持这种特性,而且提倡采用这种方式。这是一种把音效集成到项目的最佳方法。创建基于 WAV 的段的内容在第 12 章中有讲述。

A.2.4 设置乐队

前面,已经创建了自定义的鼓声循环 DLS 乐器,现在谈谈“风格”(style),特别是那些在风格模式里使用乐器。

一个乐队定义为 GM(通用 MIDI)声音集合和任何 DLS 集合的一个子集。同时,乐队也负责向 DirectMusic 传递风格中乐器的默认音量、排位(panning)、八度、变调值等参数信息。乐队中的任一种乐器必定来源于 GM 声音集合或自定义 DLS 集合,但 GM 声音集合或自定

义 DLS 集合中的每一种乐器并不一定都出现在乐队中。用程序员的术语来描述,乐队是选择乐器的附加手段。

设置一个乐队比用鼓声循环 WAV 文件创建一个 DLS 乐器更容易。需设置乐队时,双击 Band1,在 style 中的 Bands 目录中可以看到乐队编辑器,如图 A.6 所示。

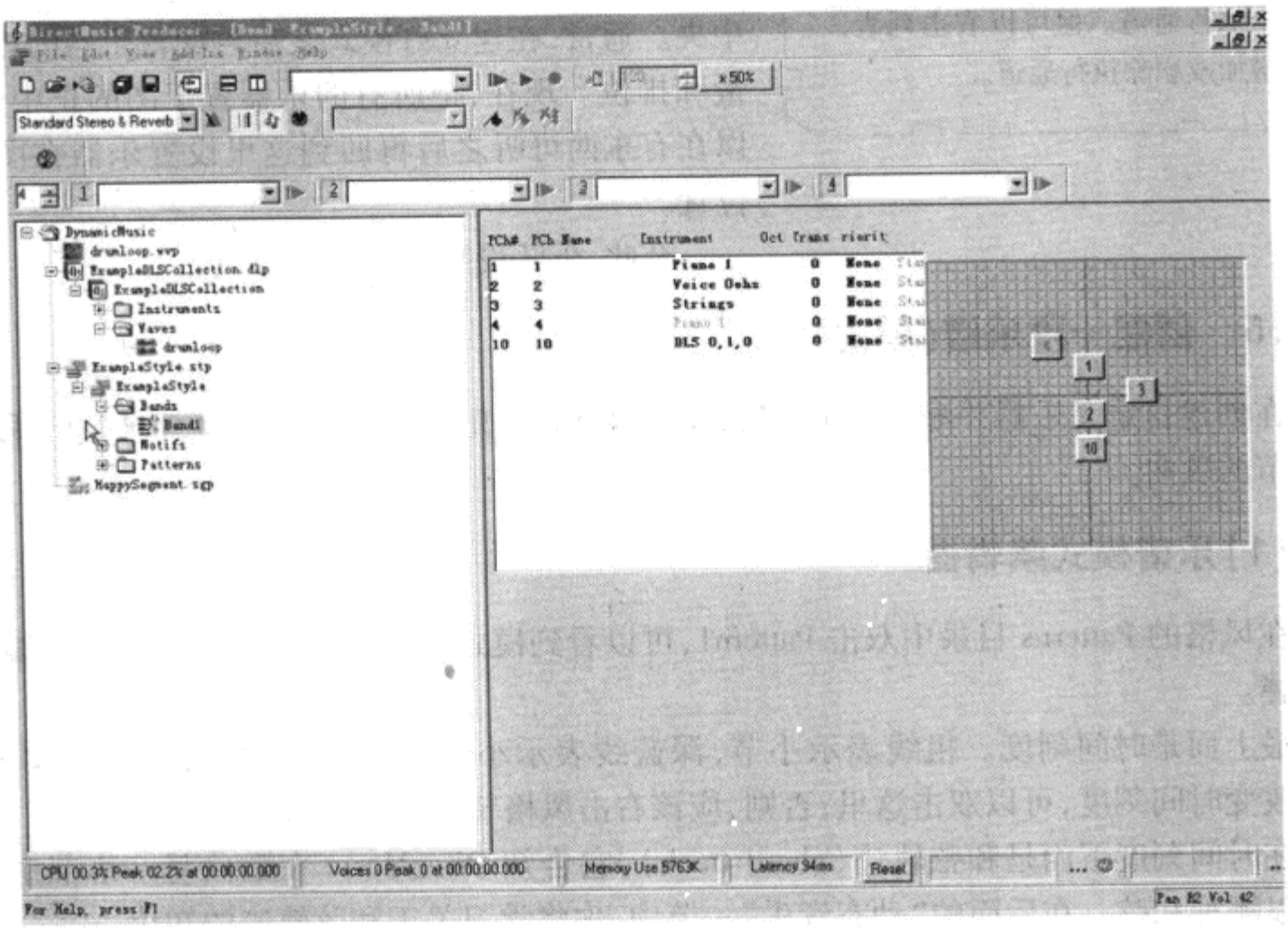


图 A.6 DirectMusic Producer 中的乐队编辑器

首先,把鼓声循环样本设定到执行通道(PChannel)10。由于 MIDI 标准要求鼓工具箱必须设定为通道 10。当在 DLS 编辑框中选择 Drums 选择框时,已告知 DirectMusic 鼓声循环是一个鼓工具箱。所以,DirectMusic 仅允许把此乐器设到执行通道 10。

在列表框左边双击执行通道 10(或者右击并选择属性)。选择 Instrument 复选框,然后点击其下的按钮。在弹出的菜单中,选择最后一项:Other DLS。然后,会弹出一个对话框,供你选择 DLS 集合及其中的一个乐器,以指定给执行通道。如果按照本附录导向完成了前述工作,你在列表中将看到(0, 10)乐器。点击 OK 以指定此乐器到执行通道 10。关闭对话框,并留心列表框发生的变化。

在示例音乐中还需要设定两个(真实)乐器。这段音乐有两个乐部:伴奏(harmony)和旋律(melody)。伴奏部分使用执行通道 1,旋律部分使用执行通道 2。由于将在伴奏中使用

琶音,建议用钢琴来演奏;至于旋律,可以选择 GM 声音集合中的“Voice Oohs”(人声合唱“哦”)。

> **提示**
>
> 在 Band Editor 中列出了 16 个执行通道。但可以右击列表,增加或删除执行通道。

图 A.6 允许调整乐器的相对音量和排位。音量由竖向位置决定,排位横向变化。如果希望一个乐器用最大音量,且处于中间排位,那么把它放到项端的中央。通常,在建立好模式之后再配置乐器的相对音量和排位。现在,把所有的乐器置于图的正中央;可以在有东西可听之后再回到这里设置乐器在图中的位置。

至此,乐队设置完成。

A.2.5 创建一段乐谱(模式)

在创建自定义乐器并配置好乐队后,就可以创建音乐了。在本小节中,将创建一段含 4 个小节的歌曲。

(1)乐谱模式编辑器

在风格的 Patterns 目录中双击 Pattern1,可以看到模式编辑器。下面是关于这个编辑器的解释。

最上面是时间刻度。粗线表示小节,深蓝线表示小节中的一拍,红线表示播放光标。如要改变时间刻度,可以双击这里;否则,应该右击风格节点,然后选择属性。

在时间刻度下面是和弦轨迹(chord track),它告知 DirecMusic 在歌曲每一小节中应分别伴以哪种和弦。在后面的“动态音乐”一节中,你将学习关于和弦轨迹的知识。

接下来是乐器的乐部。现在只有一个乐部。请注意每个乐部右端的最大、最小化按钮。点击最大化按钮,一个钢琴键盘将展现出来,可以通过它输入音符和变奏。在顶端有 32 个变奏按钮,它们位于一个 16 × 2 的表格中。在动态音乐一节中,将学习关于变奏的内容。

在变奏按钮的左面有纵向缩放按钮,窗口底部横向滚动条的右端有横向缩放按钮。表 A.2 描述了缩放的热键。

在纵向缩放按钮的左边有一个键,可以改变乐部的乐器。注意,如果这么做就等于事实上改变乐队,在这里选择一个新的乐器的效果相当于在乐队编辑器里为执行通道指定一个新的乐器。在乐器名的旁边有一个小按钮,可以打开乐队编辑器。

表 A.2 缩放热键

热键	功能
A	纵向缩小
Z	纵向放大
I	横向缩小
O	横向放大

钢琴键盘是整个模式编辑器的核心。时间横向变化、音调纵向变化、音符用竖条表示。音符越大,竖条越长,音符持续时间越长,竖条越宽。可以用鼠标单击钢琴按键以试听音符发声,也可以通过点击插入键在当前光标位置(红线)插入音符。可以用常用的编辑功能(剪切、复制、粘贴等)选择并编辑音符。

钢琴键盘的下面是变奏切换点轨迹(variation switch point track)。这个轨迹告诉 DirectMusic 在模式中间何时切换变奏是安全的。在动态音乐一节中将给出更多信息。

最下面的轨迹是控制曲线轨迹(control curve track)。它主要描述一些控制命令,比如音调降低和调制。后面将学习如何设置一条控制曲线。

(2)录制乐谱

现在开始谱曲。首先把风格里 Pattern 目录中的 pattern1 变长。打开这个模式的属性框并增加它的长度。建议先把模式调得长一些,等到录完数据后再调整到正确的长度。超长的模式允许一次录制几个乐段,而不必停止、倒回、重录。等到录了几个乐段之后,再选择最喜欢的一段,并删除其他段。最后,把剩下的一段移到模式的开始位置(可以通过剪切后粘贴、或选择后拖动实现)。

要输入音符,首先最大化每一个乐部,然后点击钢琴键盘的任意一点使其高亮化(变成浅黄色)。接下来,通过点击或按 Home 键把光标移至歌曲的开始处。最后,点击录制按钮开始录制。或者,在没有 MIDI键盘的情况下,可以先点击希望插入音符的位置,然后点击插入键以创建音符,录入音乐。

提示

录制时,如果希望计算机为自动控制时间,点击工具栏上的节拍器按钮。

现在录入如图 A.7 所示的 8 个小节的乐谱:

专业音乐人士会立即发现图 A.7 的和弦是简单的 C 调 I-VI-IV-V 进行。这即是说本模式从 C 大调开始,移到 A 小调,然后 F 大调,G 大调。记住这一点,在后续的章节中很重要。

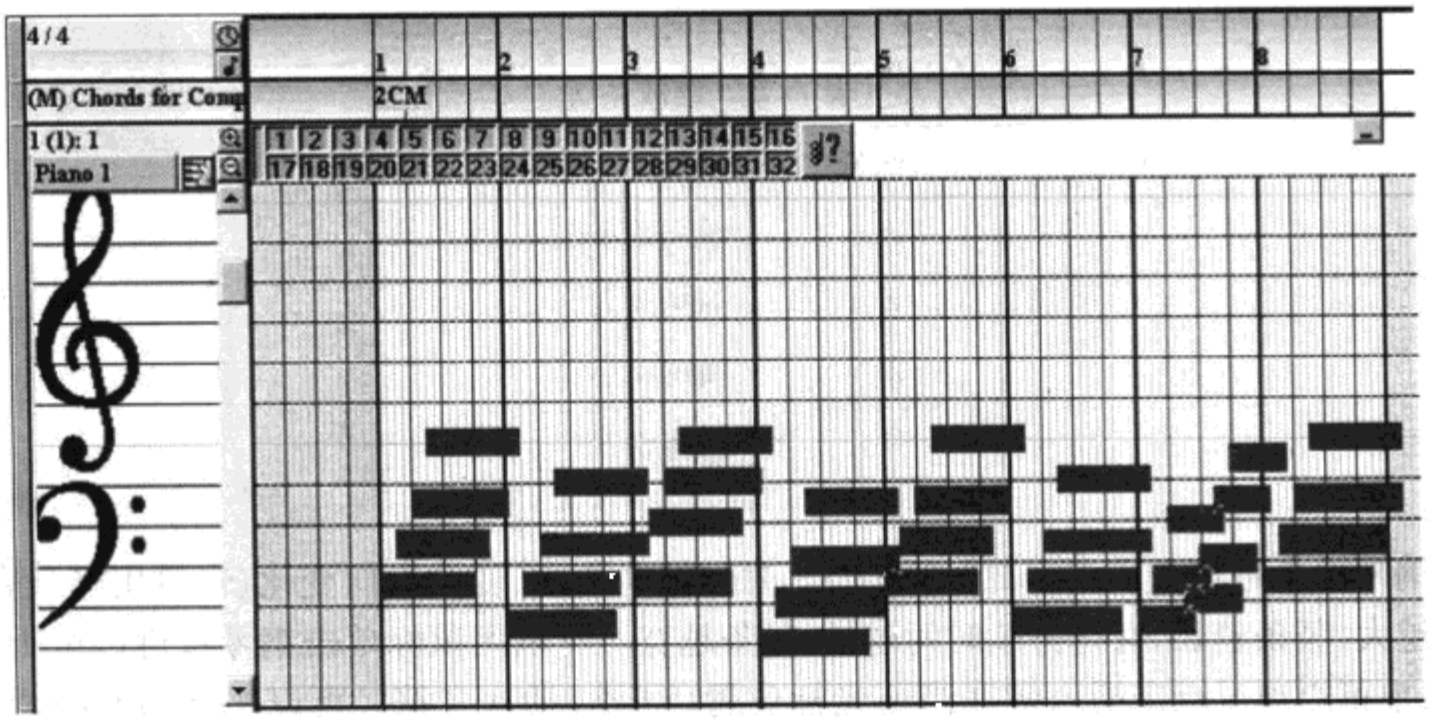


图 A.7 Pattern1 的八小节钢琴背景乐部

(3)修正乐谱

当音符输完后,读者可能会发现(除非是一个非常好的音乐家)有一些键敲早了,而另一些键却敲迟了一点。因此,不得不进行修正,以使得每个音符都按照恰当的时间排列到对应的节拍中去。如果想给音乐留下“真人演奏”的感觉,可以在如图 A.8 所示的音符属性窗口中加入一些随机的延迟(在修正好音符之后)。

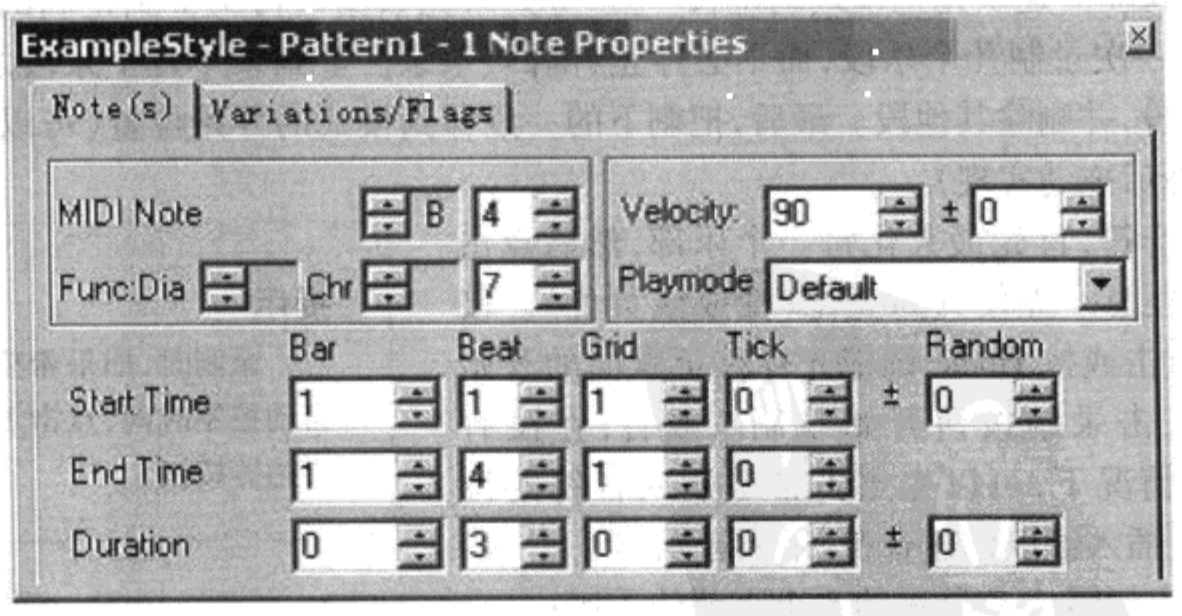


图 A.8 音符属性窗口

右击钢琴键盘,从弹出菜单中选择 Snap To None 选项。该项允许自由地移动音符。可以用鼠标直接抓着音符条的中间移动它(注意鼠标光标变化),或者直接修改音符属性框中的数字。不必试图达到完美地排列音符,差不多精确就行了。

当完成上述工作后,计算机通过一个叫做“量化”(quantization)的过程,可以精确地排列音符,使它们开始于绝对精确的时间。“量化”是指把你的音符移动到最近的一个节拍的过程。你可以选择按四分音符、八分音符、或者其他的单位进行量化。例如,如果希望用八分音符量化,点击 Edit 下拉菜单,选择 Quantize,调整所需参数,然后点击 OK 即可。

DirectMusic Producer 在量化过程中有可能会错误的布置音符,所以在排完所有音符后应该让计算机完整播放一次以确定是你想要的结果。也许需要重复好几次才能让音符的时间绝对准确。

(4)添加附加乐部

刚才录入的仅仅是背景和弦,现在需要加入旋律。首先,返回到乐队编辑器将 Voice Oohs(人声合唱“哦”,或者你最喜欢的旋律乐器)设为执行通道 2。然后,回到模式编辑器

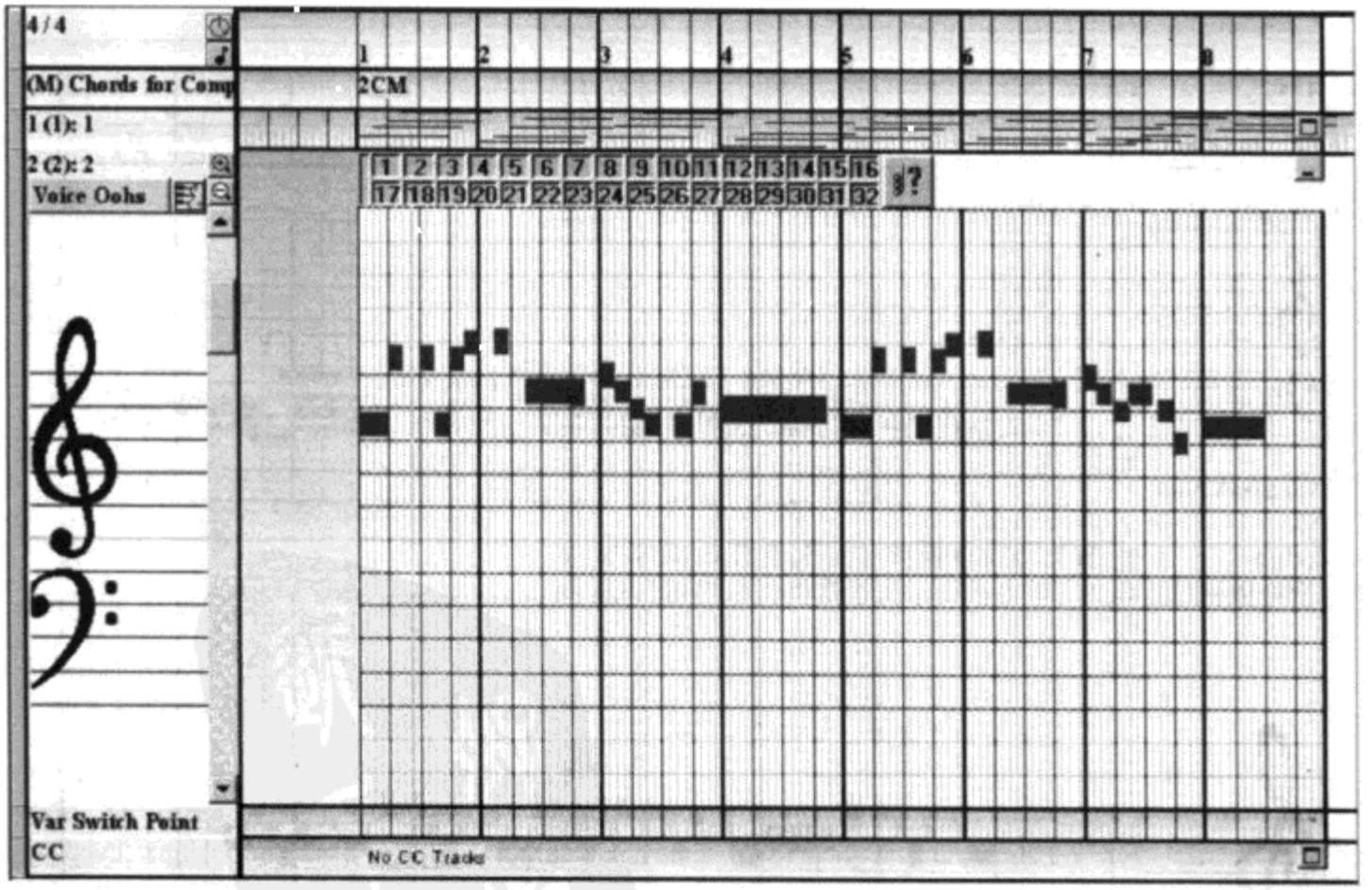


图 A.9 Pattern1 的旋律乐部

提示

另一种选择，一次性地在钢琴键盘上录入旋律与和弦。两者都录好后，再将旋律部分剪出并粘贴到第二个乐部。

并右击钢琴键盘。选择靠近弹出菜单底部的 Add Patter Parts 选项，为执行通道 2 添加一个新的乐部。

激活第二乐部的钢琴键盘并输入旋律（如图 A.9）。之后，如上一节所描述的那样量化音符。

A.2.6 创建另一段乐谱

前面，我们描述了录入第一段乐谱的详细过程。下面，再创建一个新的模式，输入两段 8 小节的乐谱，如图 A.10 所示。

完成之后，将有总共 16 小节的音乐。

提示

如果已经有一款最喜欢的可以录制 MIDI 的软件，可以就用它来录制演奏，然后把生成的 MIDI 文件导入 DirectMusic Producer 即可。在文件菜单中，选择 Import File Into Project，然后指定需导入的文件即可。

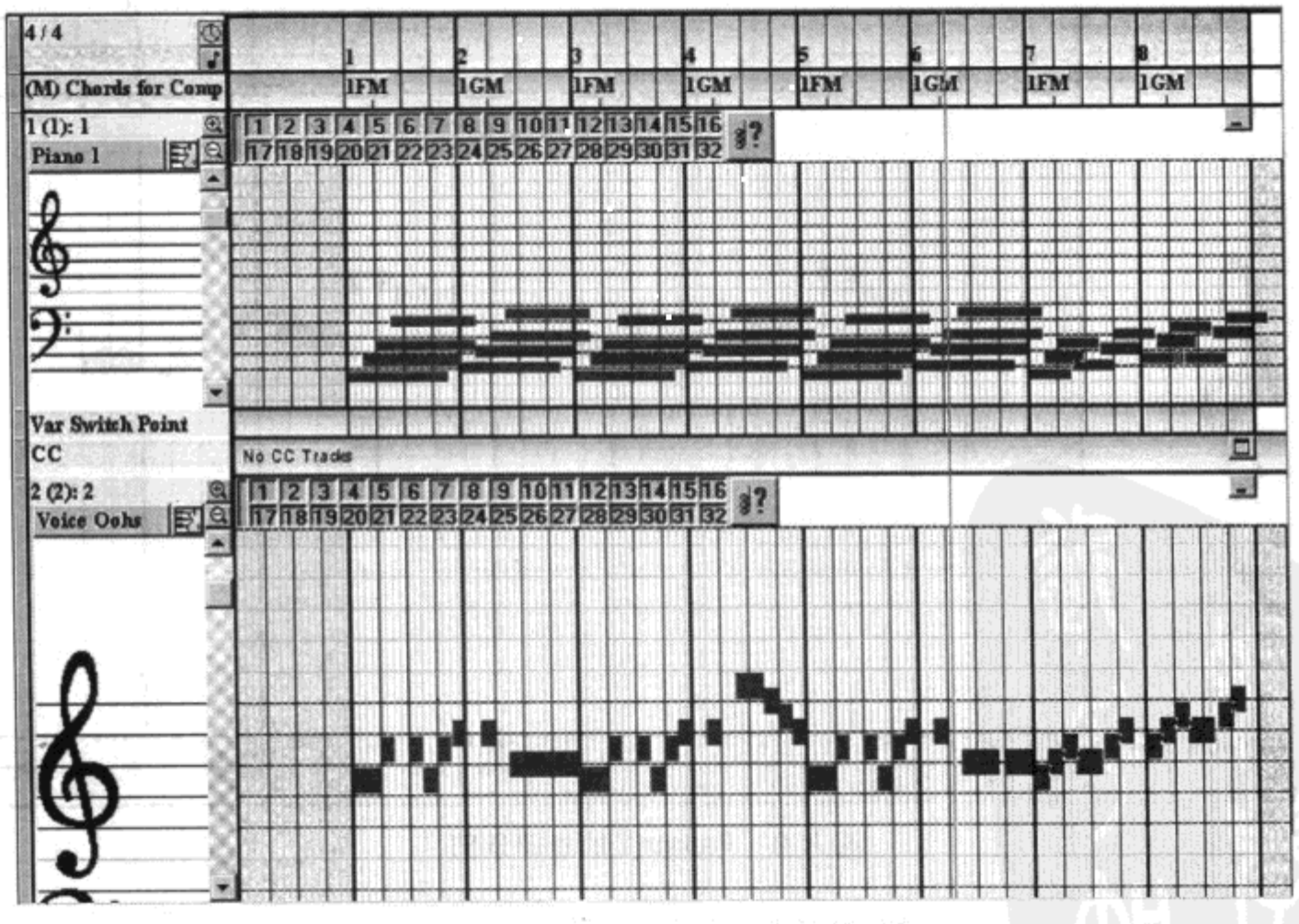


图 A.10 Pattern2 的背景及旋律乐部

A.2.7 创建段串接多种乐谱模式

创建这首简单歌曲的最后一步是把 5 个模式串起来。要做到这一点，必须创建一个含 16 小节的段。

DirectMusic Producer 会自动提醒添加轨迹。在选择默认值时，一个段只有一个时间刻度轨迹。因此，需要添加一个风格轨迹和一个强度轨迹。

风格轨迹告诉 DirectMusic 何时播放指定的风格。本例中的风格轨迹很简单——整段都播放前一节所创建的风格。把播放光标移动到段的开始处（按 Home 键），然后敲 Insert 键为风格轨迹添加一个新记录，此时将弹出风格属性对话框，选择一个风格，然后在下拉列表编辑框中找到所创建的风格，选择它，并设置其从第一小节起生效。同时，需注意当插入一个新记录时，一个新的乐队轨迹将会自动出现，且采用预设的风格。

现在 DirectMusic 将从第一小节起播放指定的风格中的某个特定模式。但它怎么知道到底是哪个模式呢？如果现在播放，将听到随机的模式演奏。在一般情况下，这样就行了。但如果希望模式按某种次序播放该怎么办呢？

这就是引入强度轨迹的原因。强度轨迹提供两个重要的参数：第一个参数设定了强度级别（groove level），DirectMusic Producer 用这个参数来排除风格中的模式（强度范围超过当前设定强度级别的模式将不会被演奏；第二个重要参数告知 DirectMusic 当有多个模式可被播放时如何处理。默认时 DirectMusic 会随机选择一个播放，可以改变设定使其按顺序播放所有的可播放的模式。

右击强度轨迹，然后选择属性。在出现的属性窗口中，选定 Repeat Patterns 下拉列表框的值为 Sequential from First。现在，播放这首歌将会听到按正确顺序演奏的模式。

A.2.8 设置段循环

最后一件事，让长 16 小节的段无限循环。在树形图中右击段并选择属性。在段属性窗口中，选择 Loop Tab 页，然后勾上 Infinite 复选框。正如读者所看到的一样，这里还有更多高级的循环参数可用，但大多数情况下，设定为无限循环就可以了。

A.2.9 将乐曲保存为运行格式

现在，读者已经可以在游戏中使用这一小段音乐了。但必须先把它保存为运行格式。这很容易，只需在文件菜单中选择 Runtime Save All，DirectMusic Producer 就会将你项目中的

> **提示**
>
> 你可以在对象中加入版权及其他各种各样的信息。在大多数对象的属性窗口中都可以找到 Info Tab 页,这个页面允许你指定主题、作者、版权及版本等信息,此外,还可以添加你所期望的任意文本。

所有文件保存为运行格式。(比源代码编译要快得多!)它会将所有的运行文件置于项目根目录下的一个名为 RuntimeFiles 的子目录里,项目根目录通常位于 My Documents\DMUSProducer。

A.3 动态音乐

前面介绍了关于如何录制乐谱、创建乐队、编辑风格及通过段连接所有元素的必要知识,现在开始为示例歌曲添加一些动态元素,通过一些基本操作使得一首歌曲每次播放时都有所不同。

A.3.1 添加变奏

让一段音乐变成动态音乐的最简单方法是给乐部加上变奏。简单讲,一个变奏就是主轨迹的一种可选替换轨迹。

给乐部添加变奏的操作通过位于钢琴键盘上方的 32 个变奏按钮来实现。每个乐部可以有 32 个变奏,32 个变奏按钮按每行 16 个排成两行。每个按钮传递 3 个至关紧要的信息:变奏是否处于激活状态,变奏是否可用(enabled)及它是否包含音符。

先解释最简单的,如果一个变奏包含音符,其显示数字将为黑色;如果变奏不包含音符,显示数字将为白色。

其次,如果按钮是下陷的,变奏处于激活状态。即它的音符显示在钢琴键盘上,且可以编辑。

最后,一个按钮的背景色告诉你它是可用的或者不可用的。浅背景色表示可用,深背景色表示不可用。注意,变奏是否可用与是否处于激活状态的含意是不同的。一个可用的变奏可以在播放过程中随机地选择启用;而 DirectMusic 永远不会选择不可用的变奏。

默认情况下,乐部所有的变奏都是处于激活状态且可用的。当你刚开始新创建一个模式(一段乐谱)时,可以注意到所有的变奏按钮上的数字都是白色的,这意味着它们都不包含音符。当你输入第一个音符后,它们都将变为黑色,这意味着你是在同时编辑 32 个变奏。这样做使得你可以同时将基本旋律录入 32 个变奏中,然后再一个个调整。

现在开始操作。首先,单击 2 号变奏按钮(1 号变奏将是最初的音,不改变旋律),它将弹起,表示你现在输入的任何音符将会录入到除 2 号变奏外的所有变奏。如果这并不是你

想要的！你希望翻转所有的按钮，请双击 2 号变奏按钮。现在除 2 号按钮外，其他按钮都将弹起来(即不活动)。

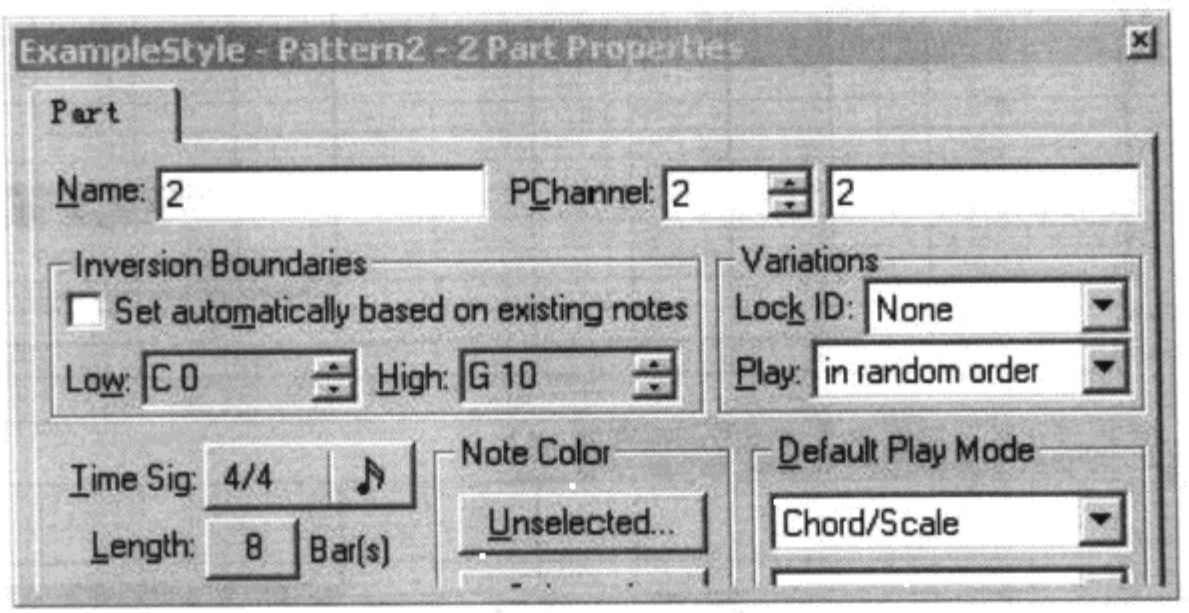


图 A.11 乐部属性窗口

给乐部添加适当的变奏会让你的乐曲更加有趣。你可能希望转位琶音、添加一个修饰音(frill)、做一个切分(syncopate)或做其他符合你心愿的任何事情。在你设置完 2 号变奏后，可以接着设置变奏 3 等等。通常两三个变奏就很好了，很少使用到全部 32 个变奏。

完成变奏之后，再播放你的乐谱(模式)，这时，DirectMusic 将随机地选择变奏。如果希望换种方式(比如按顺序播放变奏)，你可以通过乐部属性窗口(如图 A.11)给变奏指定顺序。设置乐部属性，可以在钢琴键盘上右击除音符处以外的任意地方，然后选择属性。在 Variations 组里，有一个名为 Play 的下拉列表框，允许你选择不同的行为模式(这些行为模式类似于强度级别行为模式，参见 DirectMusic Producer 的帮助文档以寻找完整的解释)。

A.3.2 添加修饰

每一种修饰实际上也是一种特殊类型的模式，主要有过门(intro)、间奏(fill)、暂停(break)或尾声(end) 模式。

首先介绍如何创建一段“过门”，然后将其设定为在段的开始处播放。

创建“过门”，需首先建立一个新模式，并打开其属性窗口，点击 intro 按钮，告知 DirectMusic 这个模式是一段“过门”。然后，创建一段很酷的过门(如图 A.12 所示)，并将其命名为“Intro”。

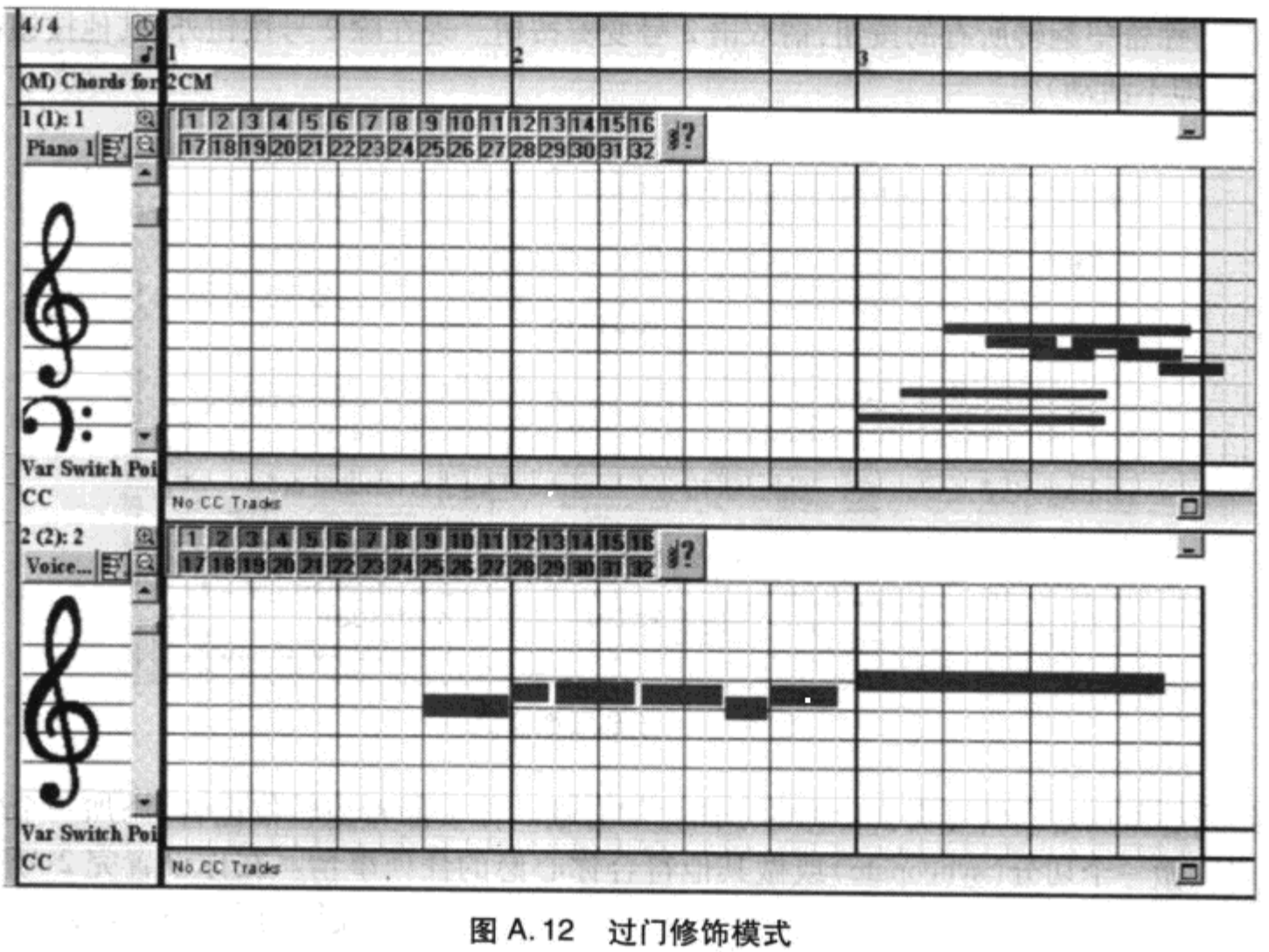


图 A.12 过门修饰模式

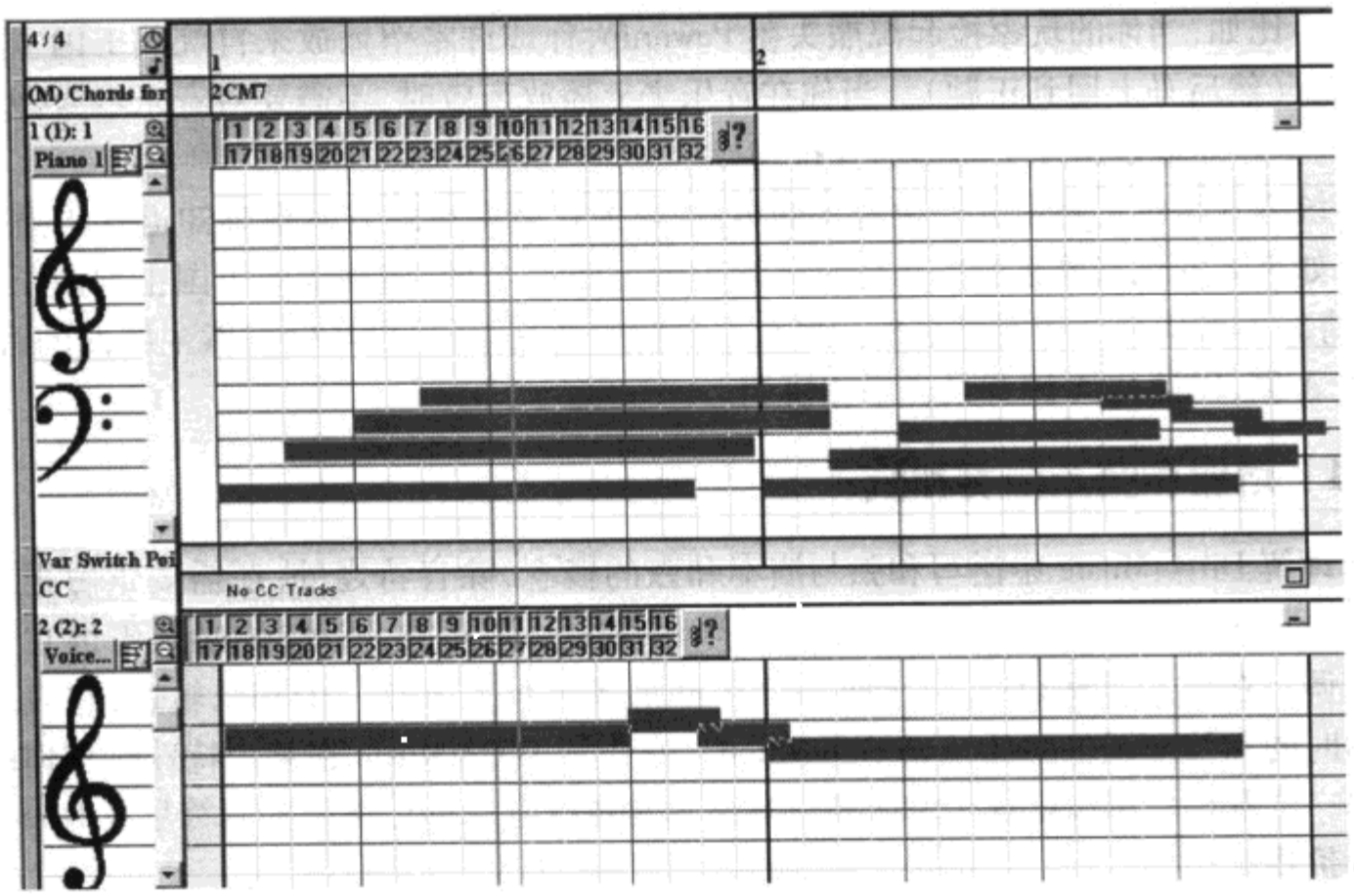
接下来,我们用同样的方法创建一个“暂停”模式。录制如图 A.13 所示的一小节乐曲,将它标记为暂停模式,并将其命名为“Break”。

现在,需要设置一个段去播放过门修饰及暂停修饰。在树形视图中右击你的段,并选择属性。修改其长度为 23 小节,然后将 Loop Tab 页中改 Loop 从第 4 小节开始,22 小节结束。请确定无限循环(infinite)复选框被选中,这样你的段才会永远循环。

设置完成后,你的段的长度已经足以存放过门及暂停小节了。在强度轨迹的起始记录处右击,然后选择属性。在属性窗口中,把 Embellishment 下拉列表框中的 Intro 记录高亮化。把强度级别设为 1,这将告知 DirectMusic 它可以播放任何强度级别为 1 的过门模式。

现在,打开 Pattern1 的属性窗口,设置其强度范围为 1-1。这将使得仅当强度级别设为 1 时,DirecMusic 才有可能播放此模式。用同样的方法设定模式 2 的强度级别为 2-2。

回到段编辑窗。由于你的过门有 3 个小节,把红色光标竖线拉到第 4 小节的开始处,然后按 Insert 键。现在,确定 Embellishment 下拉列表框中没有行被选定,并保证强度级别仍为 1。同时,也确认 Repeat Pattern 下拉列表框设定为 Sequential from First。这些参数告知 DirectMusic 从第 4 小节开始,它可以播放强度级别为 1 的任意非修饰模式(本例中仅有 Pattern1 满足此条件)。



A.13 暂停修饰模式

接下来,在强度轨迹中跳到第 12 小节,插入另一个记录,并设定其强度级别为 2。这将强制 DirectMusic 播放 Pattern2(Pattern2 是强度级别为 2 的惟一一个模式)。

最后,在强度轨迹中跳到第 20 小节,插入另一个记录。这里,设定强度级别为 1 并在 Embellishment 下拉列表框中选择 Break。

完成上述设置后,再播放这个段,你将先听到过门,接下来是 Pattern1 及 Pattern2,最后是暂停模式。在暂停模式之后,DirectMusic 将循环并重新播放 Pattern1(注意到循环范围将过门模式排除在外,因为你只希望它播放一次)。

如上所述,创建一个修饰就如同创建一个模式或者点击一个按钮,非常简单。

A.3.3 添加乐旨

动态音乐之所以酷,不仅仅因为它是动态的,还因为它可以对游戏事件做出反应。动态音乐可以提供两种类型的反应。

首先,如果有重要的事件发生,你可以完全改变歌曲。例如,如果你的玩家遭到一大群敌人的袭击,动态音乐可以迅速地增强强度;当玩家将敌人全部打败时,它可以慢慢缓和下来,最后停顿。这是一种典型的反应方式,但在学习“创建可选段”一节之前,你还不知道如何应用这种反应方式。

动态音乐的第二种反应方式是在有趣事件发生时,短暂地增强或者发出一小段迅速的叮当声。比如,当你的玩家捡起浣熊头盔 Powerup,你或许希望播放来自浣熊王国主题曲的几个音符(然后马上回到正题)。当你在音乐之上播放声效时,不需要对音乐本身做太多的改变;一个声效就是由几个简单的音符构成的。

传统方式很难创建适合一段音乐的声效,因为在事件被触发时很难知道音乐播放到哪里了。如果你的玩家捡起浣熊头盔时背景音乐正处于 A 调,而且浣熊王国主题曲正好是 B 调,你肯定不希望把二者混合起来。那么需要做的就是编码,以降低浣熊主题曲的音调,使其位于 A 调,以便与 A 调背景音乐相协调。

A.3.4 谱曲和弦及演奏和弦

要理解 DirectMusic 中谱写和弦与演奏和弦的概念,除音符之外,还需要学习更多的知识。DirectMusic 不能自动推断出你的音乐采用哪个基调播放,因此,你必须告诉它关于乐曲的基调。在模式编辑器中,位于每一个乐部钢琴键盘上方有一个和弦轨迹,它表示谱写该乐曲时的和弦基调(chords for composition),它告知 DirectMusic 在某个特殊时刻你的音乐处在哪一个基调。可以把这个和弦轨迹理解为是传统音乐标记法中基调符号在 DirectMusic 中的等价物。

然而,用 C 调谱写的乐曲(模式)并不一定意味着它正以 C 调被演奏。我们可以借助段演奏中的和弦轨迹,让 DirectMusic 为你变调。例如,可以创建一个模式并设定整个模式的谱曲和弦轨迹为 CM7(C 大调、第 7 音阶),然后,在段中创建一个演奏和弦轨迹并设为 DM7(D 大调,第 7 音阶)。当 DirectMusic 在段的演奏和弦轨迹中遇到 DM7 时,会想到:“这段乐曲是用 CM7 写的,但现在需要用 DM7 播放,所以我将它提高两个半音。”如果用数学化语言来描述,可以想象成 DirectMusic 将两个声调值相减,然后把差值应用于你的乐曲。

正是由于这种原因,如果你希望一段音乐按照它被谱写时指定的基调演奏,你就必须保证这段乐谱(模式)的谱曲和弦基调与段演奏和弦基调是相等的。即保证谱写调和弦与演奏和弦调的差值为 0,DirectMusic 播放时就不会改变歌曲的谱写时的基调。

(1)输入乐旨

前面描述的是原理。现在简述其实现方法。对于一个初学者来说,先创建一个新的乐旨(右击 Motifs 目录,然后选择 New,并输入如图 A.14 的乐旨)。

正如读者所看到的,这个过程并不复杂:仅仅是一些快速播放的音符,它们听起来可能更像声音而不是一小节音乐。

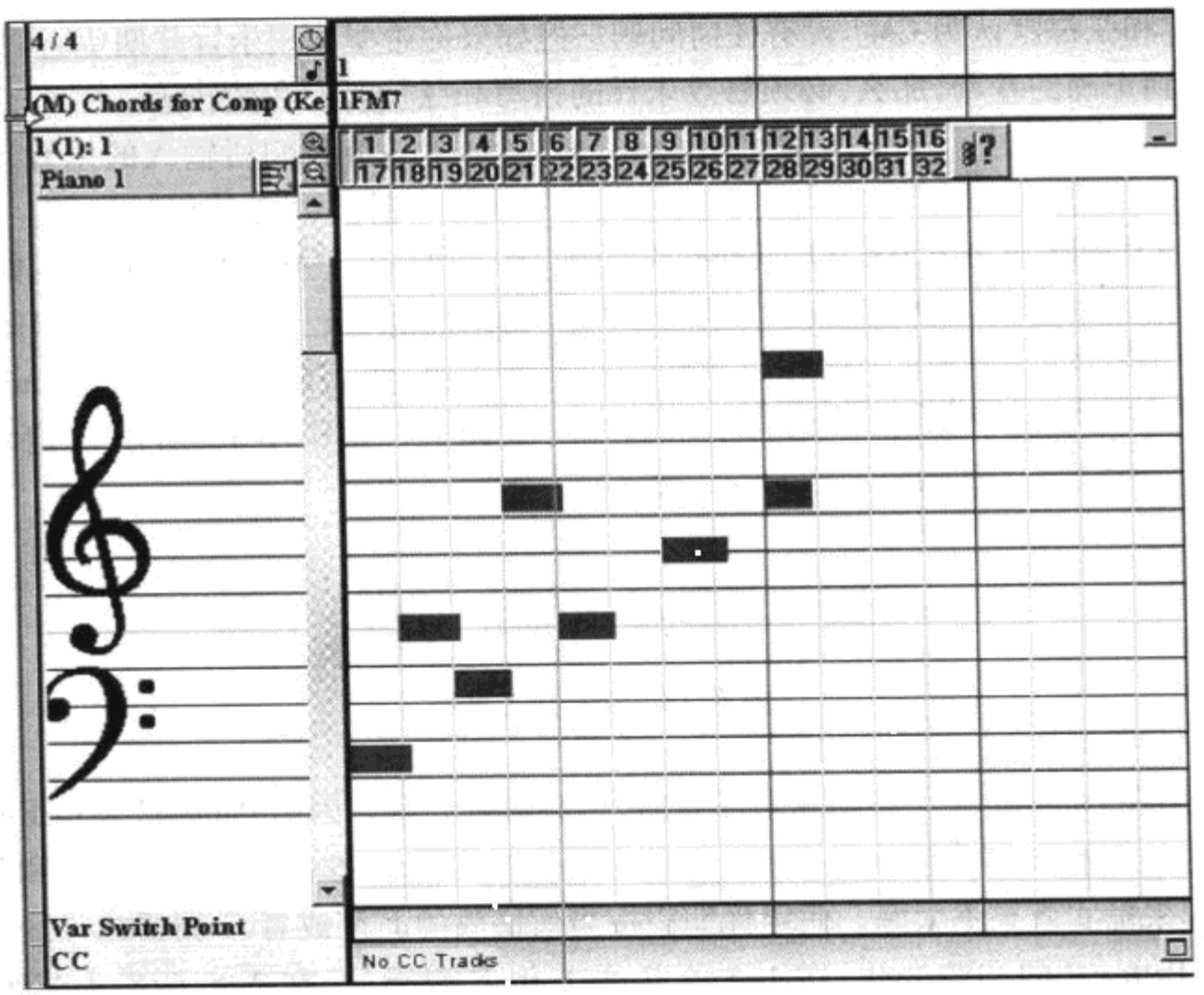


图 A.14 一个简单的乐旨模式

这里,通过右击和弦按纽并选择 Add Tracks,可以给段添加和弦轨迹(乐曲的基调)。你会看到一个新的和弦轨迹将出现在段编辑器中,其默认值为 2CM7。

提示

乐旨是一种有特殊标记的模式,它与一般模式一样,可以有多个乐部及变奏。前面已经学会了如何添加变奏,因此给乐旨添加变奏就很轻松了。变奏将使你的音乐具有动态特性。

(2)设置和弦

DirectMusic 播放时如何进行模式转换取决于谱写和弦与当前段和弦的差异。对乐旨而言道理是一样的。当你的代码命令 DirectMusic 播放一个特定的乐旨时,DirectMusic 会观察乐旨的谱写和弦与当前段和弦轨迹的差异。如果乐旨是 EM7,而当前段按 GM7 播放,则 DirectMusic 会将乐旨提高三个半音,让其播放于 GM7,使其与当前段和谐。

到本示例的目前为止,你还没有更改任何谱写和弦或者段和弦轨迹。所以,所有的和弦轨迹都被默认设为 CM7。因而,演奏时不会发生任何转换。如果你不准备使用乐旨,那

么,让所有和弦都默认为 CM7 没有任何问题。但如果你希望使用乐旨并期望它们在播放时自动转换到正确的基调,那么,必须修改乐旨的谱写和弦及段的和弦轨迹。

提示

当你试验时,注意不论什么时间敲播放键,DirectMusic 总会在一个节拍的边界处开始播放乐旨。当然,这是可以调整的,但在大多数情况下,把乐旨排列在节拍边界效果会很好。

上一节我们所输入的乐旨,其基调为 CM7,不需要改变其谱写和弦。这个乐旨在 Pattern1 上播放将会听到良好的效果。如果你想自己试一试,请先启动 Pattern1 的播放,然后通过次(the secondary)播放工具条(如图 A.15 所示),启动乐旨的播放。

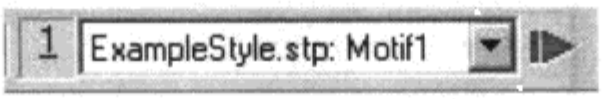


图 A.15 次播放工具条

当试图在 Pattern2 之上播放这段乐旨时,将会遇到问题。因为 Pattern2 是按 F-G-F-G 调进行的,这与你的 C 调乐旨并不协调。尽管它不会像在 F 调之上播放一个 B 调和弦那样糟糕,但听起来确实效果不太好。假定你期望按当前播放的 F 调或者 G 调播放你的乐旨。即期望它随 Pattern2 的和弦变化,在 1、3、5、7 小节用 F 调播放,在 2、4、6、8 小节则用 G 调播放。

为了完成这些设置,请打开 Pattern2 并注意其谱写和弦轨迹。你可以看到第 1 小节使用的是 2CM7 基调。右击它并选择属性,它将引导你进入和弦属性设置窗口,如图 A.16 所示。在这个窗口里,需要做两件事。

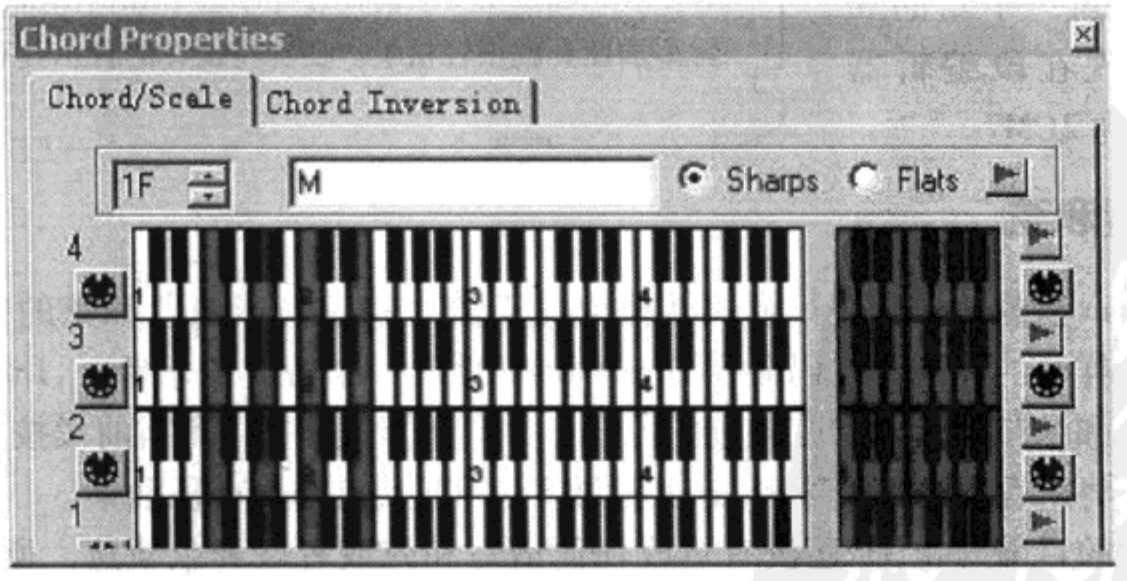


图 A.16 和弦属性窗口

首先,你需要点击“下调按钮”7 次,把基调值从 2C 降低到 1F。注意,当这样操作时,红色的音调键将随之移动。你必须保证每个频道(1-4)的 4 个红色音符构成一个 F 主和弦(F-A-C-F)。可以通过单击琴键去调整它们。

提示

当改变红色钢琴键音符时,从最低通道,通道 1 开始。当你改变通道 1 的红色音符时,DirectMusic 会自动改变通道 2,3,4 的红色音符。通常,你都会期望 4 个通道具有同一个红色键,所以这是条很好的捷径。

请注意显示为 M7 的文本框,用音乐术语表述,这个文本框显示的是你当前正通过钢琴键输入的和弦。DirectMusic 并不在意这个文本框显示的内容,它是为人类音乐家准备的。

和弦属性窗口还可以做很多奇妙的事情,但对本例而言,我们所做的已经足够了。关闭和弦属性窗口,然后把光标线(竖向红线)移到下一个小节的开始位置。当谱写和弦轨迹被高亮化(浅黄色)后,敲 Insert 键,然后用同样的方法在和弦属性窗口中输入 1GM7 作为本小节的谱写和弦。像这样依次处理后续的每一个小节,1FM7 和 1GM7 交替循环,需要设置 Pattern2 及 Pattern2HighGroove 的所有小节。

你也许希望在每个值中输入 M 而不是 M7,确定所有的红色音符构成一个主和弦,而不是主调第七和弦。这只需要打开所有的段和模式的和弦属性窗口,并将所有的 2CM7 值改为 2CM。

等一等,现在还没有做完。如果现在播放你的段,你会发现 DirectMusic 将转换 Pattern2 及 Pattern2HighGroove,声音听起来会很可怕。DirectMusic 之所以转换它们的原因在于你仅改变了 Pattern2 的谱写和弦,还没有改变段的和弦轨迹。你需要给段的和弦轨迹设定合适的值以同模式相配合。这意味着,应该设定第 1 小节及第 20 小节的和弦值为 2CM,第 12、14、16、18 小节为 1FM,第 13、15、17、19 小节为 1GM。这样设定之后,段的和弦值将始终等于模式(乐曲)的谱写和弦,所以,当 DirectMusic 播放时就不会做转换。

现在再做一次最后的测试。在段播放的任意时间播放乐旨,你会注意到 DirectMusic 已经自动地根据段和弦轨迹将乐旨转换到 F 或 G 调。声音很酷,是不是?

当然,你可以综合运用和弦、模式及乐旨,使乐曲演示效果的更加复杂,也更加有趣。DirectMusic 可以在任意两个和弦之间转换。例如,它可以简单地从大调转换到小调——你惟一需要做的就是在和弦属性窗口中正确设定红色钢琴键。请自己实验一下吧!

(3)关于和弦路径

结束本节前,需要特别说明的是 DirectMusic 还具有另外一个重要特性。实际上,你并不需要通过手工方法一小节、一小节地依次设置和弦,因为 DirectMusic 支持自动选择和弦。

如果模式支持并使用得当,可以创建出难以置信的动态音乐。而你所需要做的事情就是设置一张“和弦地图”(chord map),以告知 DirectMusic 在段中它可以选择哪些和弦。DirectMusic 将自动地从和弦地图中选择一条路径,让你获得真正动态音乐的效果。

关于和弦路径的更多信息,请阅读 DirectMusic Producer 提供的简介。

A.3.5 创建可选段

本示例最后一件事情是为高强度级别创建可选模式段。这样,游戏可以控制音乐的强度。通过代码修改强度级别编辑器中的设置,你可以控制强度级别,强度级别编辑器的默认值为 0。最终的强度级别取决于强度级别编辑器的设定值和由段决定的当前乐曲的强度级别值之和,它最终决定模式(乐曲)的播放强度。

我们已经创建了两个模式(乐曲),其强度级别分别为 1 和 2。现在创建两个强度级别为 11 和 12 的新模式,取名为 Pattern1HighGroove 和 Pattern2HighGroove。这两个新模式包含了与 Pattern1 和 Pattern2 相同的音乐,只是多了一个鼓声循环,使其效果更强烈一些。从这里可以看到一个基本思想:当某游戏事件发生时,你可以通过程序修改强度级别,如设置编辑器的值为 10,以便播放强度级别更高、效果更强烈的乐曲。

添加一个鼓声循环并不难。先通过剪贴板复制 Pattern1,另命名为 Pattern1HighGroove,然后添加一个新乐部,并指定其乐器为自定义鼓声循环。接着,为乐部添加一个大音符 C5,并持续 4 个小节。在第 4 小节,再创建另外一个 4 小节长的音符,这样你的鼓声循环将一直延续至后 4 个小节。

用同样的方法处理 Pattern2,以创建名为 Pattern2HighGroove 的拷贝。不要忘记到模式属性中修改其强度级别为 11 和 12。

A.3.6 保存动态音乐

完成了各种设置后,不要忘记了把你的动态音乐保存为运行格式,并放置到第 11 章源代码所在的目录中。

A.4 小结

创建一段优美的动态音乐需要学习的东西很多。在本附录中,主要学习了最基本的概念和设置方法:变奏、修饰、乐旨、可选段、和弦映射等。还有一些不常用的概念本附录没有涉及。

如果你希望成为一个动态音乐专家,就需要花费大量的时间来创建动态乐谱,并不断调整,直到你已掌握了 DirectMusic Producer 的全部特性为止。DirectMusic Producer 是一个

功能复杂的实用软件，需要慢慢学习并多多练习。

请记住，书写动态音乐比书写静态音乐要难，很容易为细节花费太多的时间。如果游戏中有一个分支需要 30 秒钟的音乐，而玩家只有一种可能的情况会进入该分支，那么，你就没有必要浪费时间，为那段音乐添加太多的变奏和可选段等动态效果。通常，玩家应用越频繁的音乐，开发者就应该花费越多的时间来设计动态效果。

PDG

附录 B

创建游戏音效的体会和技巧

如果在游戏设计中你同时担当程序员和声效设计师的角色，那么这个附录可能对你有所帮助。我整理了一些为游戏创建高水平声效和音乐的一些体会，现在将它们归纳如下，希望它们能够给你提供一些帮助和参考。

B.1 音乐

游戏音乐常常是游戏研发中最容易被忽视的部分。人们通常认为：游戏可视化部分比游戏音效更重要，而且在游戏音效中，游戏的声效和语音比游戏音乐更重要。我要强调的是：千万不要让这种事情在你的游戏中发生！音乐是你在游戏中传递情感的最强有力的工具之一。只有图像效果的游戏受欢迎的程度绝对不会太高，配有优美音乐的游戏通常比那些图像效果一流但音乐效果糟糕（或者没有音乐）的游戏更受欢迎。

下面介绍为游戏创建高水平音乐的几个体会和技巧。

B.1.1 创建音乐并不需要音乐家

要创建好的游戏音乐，并不需要成为专业音乐家。在图像处理领域，即使你没有艺术方面的修养也没有关系，借助图象软件，可以帮助你创建非常理想的图像效果。如 Bryce、PhotoShop、Kai's Power Tools、Vue d'Esprit 有时可以把你的游戏从“程序员艺术”陷阱中拯救出来。类似的，也有很多软件可以帮助你快速容易地创建高水平的音乐。本书配套 CD 中提供了一些软件的演示版及相关的链接。

B.1.2 游戏中引入畅销歌曲

在游戏中引入包含畅销歌曲是很重要的。在将歌曲收录到游戏中之前，你必须明白：这样做需要花钱买版权许可证。

从技术角度上讲，将畅销歌曲加入游戏非常容易。找到并打开 MP3 收藏，然后将歌曲加入就可以了。然而，在没有获得歌曲版权许可之前，分发含 MP3 的游戏将违反版权法，并面临数十万美元的诉讼。即使你不准备通过游戏赚钱，在没有获得授权的情况下分发其他人的歌曲也是违法的。

将畅销歌曲收入游戏的合法途径是与其所有者联系，并告诉他你的打算。在联络之前

要准备好相关细节，比如你是希望使用整首歌还是它的某个片断。

B.1.3 招聘懂乐理知识的人员

如果你认识一个乐队成员或者正准备组建乐队的朋友，如果你知道一个乐队的歌曲与你的游戏非常契合，则可以考虑同他们做个交易。这样，你将不必亲自书写和录制游戏音乐，而他们也将获得急需的激发他们创作热情的知名度。当你被一个 Bug 折磨数天并几乎想到放弃整个项目并重新开始时，如果有除你之外的其他人在盼望着游戏的发布，将会是很有帮助的。

对于乐队，他们应该“从对游戏的贡献中获得何种收益”没有固定的规则。报酬有可能是 6 听饮料或者数千美金，或者是在游戏开始处突出“Music By”的提示，也可能是游戏利润的一部分。你需要自行谈判，结果应该是你所能提供的且乐队所能接受的。例如，如果你从未发布过一个游戏，向乐队承诺一定的版税比例可能并没有多大意义。对于某些乐队这可能足够了，但对于另一些乐队，他们可能还希望得到一些现金。

当你选择好乐队后，就需要向他们详细描述你到底期望什么。对于你所期望的每一首歌，你都需要向他们解释你所期望表达的情感、音乐所处的环境、长度以及歌曲本身是否需要循环点以便于你的代码无限制地播放它。通常情况下，游戏音乐都没有歌词，它更像是电影或电视里的音乐，目的是为了烘托气氛。然而，在有些情况下，有歌词的音乐也不错，你可以借鉴电影里的做法。

总之，与一个懂乐理知识的朋友一起工作，对双方都是有益的。你们将会从对方吸取创作的灵感（他或者她会给你的游戏出很好的主意！）。

B.2 声效

作为对玩家动作的反应，你可以播放一个 WAV 文件，但播放哪一个好呢？这是本节需要讲解的内容。这里，你将了解创建高水平游戏音效的一些技巧。

B.2.1 何时播放声效

在游戏开发中，最常见的问题之一是通常所称的“不平衡声效”（lopsided sound effects）。不平衡声效意味着游戏给那些无关紧要的动作添加了声效，但却往往忽略了最重要的动作。例如，假设有一个幻想动作游戏，其中有一个挥剑的英雄和几个愤怒的小魔鬼。我曾经见过一些业余游戏为小魔鬼的咕哝、受伤、死亡设置了几种不同的声效，但却惟独没有舞剑声、玩家受伤或者剑击的声音。

为避免“不平衡声效”的发生，必须遵循几个简单的声效原则。对于初学者而言，要记

住声效是游戏告知玩家一些事件发生的最重要的方法。它们通过声效向玩家传递信息。这个效果达到之后,声效还有另外一项功能,即使游戏具有情感和沉醉感。

一个原则是确定“玩家在游戏中的任何一个动作都有声效”。另一个原则是保证“即使一个人站在看不到计算机屏幕的地方,他也可以通过声效知道玩家正在做什么事”。

B.2.2 利用声效集

我们已经知道在游戏中应当包含什么声效,以及应该在何时触发每一种声效,接下来,应该考虑怎样创建这些声效。

其实,完全没有必要自行创建所有需要的声效。市场上和网站上有很多可以剪辑应用的“声效集”,可供你购买和使用,而且在游戏应用中使用这些可剪辑的声效资源是不需支付版税的。这些声效集大多数都提供高质量立体声的 WAV 文件,你可以降低其采样频率并转换成单声道,以方便在游戏中使用。

声效集的价格差异很大。专门为专业电影和广播设计的声效集很贵,但也有一些声效集的价格与一套视频游戏差不多,甚至只相当于二手市场上的一套游戏的价格。

一些声效集还包含了一些免版税的音乐片段,对于游戏结束、游戏选项这样一些你仅需要小段音乐的场合,这些音乐片段非常合适。

本书附带的 CD 中,提供了一些销售声效集的网站链接。

B.2.3 自己录制声效

如果你找遍了声效集都无法为那扇吱吱响的门和那个打鼾的小妖精找到合适的声效,那就需要自行录制。这里是一些关于如何录制声效的小技巧。

- 最小化背景噪声:只要条件允许,尽可能将麦克风远离计算机及其轰轰转的风扇,或者使用一个可移动的录音机。同时,找一个有较好声学效果的房间。有时,浴室是个好地方,因为瓷砖可以很好地反射声音。
- 声音录制后可修改:没有任何规定说你不能把一个人的打鼾声的音调调低,以创建一个食人鬼的鼾声效果。你可以用声音软件改变声音的各种特性并播放:倒转声音、改变音调、应用滑音(flange)或者 filter 特效等。
- 逆向思维:不要这样考虑:“因为我需要一个吱吱响的门的声效,所以我要找一个声音像吱吱响的门的物体”。相反,应该随便找一些物体,然后闭上眼睛,想想“这个声音像不像吱吱响的门? 或者能否把它改造成门吱吱响的效果?”,游戏的可视化图像将会使声效的效果听起来更加真实。在现实世界中,也许某个声效就是猫的叫声,但在游戏中,当玩家打开一扇门时,这个叫声听起来就更像是开门的吱吱声了。由于玩家判断声效的意义并不仅基于声效本身。

- 请求帮助:“声效搜寻”有时会非常有趣,获取“从雪中走过”声效的方法是爬上雪山,并录制某个人从雪中走过的声音。呼朋唤友,去探险吧!

B.2.4 不要从网上下载声效

尽管从网上下载声效非常简单,但风险太高,因为网上的 WAV 大多数都是从电影、电视或者其他视频游戏中摘录出来的,这意味着使用它们与使用没有付费的畅销歌曲一样,是违法的。既使你对它们作了表面的修改,你仍然属于“在未获得原创者同意的情况下重新发布这些声效”。

很难判定一个来自网上的声效是否是免费的。即使一个网站声称某个声效是免费的,但如果这个声明同辛普森案中的“D'oh!”很相似呢? 那最终当你发布你的游戏之后,就会收到来自 FOX 团队的信,他们想知道为什么他们角色的声音出现在你的游戏中。

B.3 语音

语音(speech)是游戏音频的一个重要组成部分。早期的游戏中并不需要语音,因为添加语音被认为过于前沿,而且对于游戏玩家的经验而言,把语音作为游戏的一个集成部分还不太习惯。但在今天,即使游戏配有不太好的语音扮音,也比没有配音好。

本节介绍如何为游戏集成高质量的语音。

B.3.1 聘用配音人员

有很多专业的、充满热情的配音人员愿意为你的游戏角色配音。聘用配音人员通常不会太贵。当然,报酬开得越高,聘用的配音人员素质就越高。

与专业配音人员一起工作,在技术上是非常容易的。因为他们几乎所有人都有专门的录音工作室,并懂得如何通过电子邮件来传递声音文件。大多数情况下,你只需把男女配音人员要配音的剧本片断发给他们即可。解释角色在某一点的心理状态也会有所帮助。

如果你不准备花钱聘用专业配音人员,可以考虑请求朋友的帮助。同你所认识的人交谈,把脚本递给他们,设置好录音设备,然后请他们一句接一句地朗读他们的台词。很多人喜欢配音,特别是当他们知道他们的肖像不会随语音一起发布的时候。当然,自己也可以试试配音。当你不再为听到自己录制的声音感到惊奇时,就可以为游戏配音了。这非常有趣。

B.3.2 声音本地化

声音本地化是处理游戏语音的难题之一。如果你想让那些听不懂英语的玩家可以很

容易地玩你的游戏,那就必须进行声音本地化,如声音汉化。

本地化会大大增加配音人员的工作量。这意味着你需要聘用会讲不同语言的配音人员。

要仔细思考你的游戏中需要包括什么语言,以及这些语言是否都需要独立的语音文件。如果声音对话对于游戏进程并非十分重要,你可以仍让扬声器讲英文,而将对应语言的文本显示在屏幕上。

B.3.3 声音与口形同步

处理游戏语音的另一个难题是让角色的口形与扬声器的发声同步。即使今天,大多数的游戏仍没能很好地完成这项工作,以使得玩家相信他们的角色正在说话。然而,随着技术的进步,现在市场上已经有专门用于处理游戏或者计算机渲染中角色口形同步问题的软件。

要正确地对上口形,需要将源声音分解成单个的音素。音素和语音的关系就如同比特(位)和数据的关系,换句话说,音素是构成语音的最小组成单位。每一个不同的音素都可以映射到角色的不同口形,这样,如果你将语音拆分成单个的音素,就很容易对上角色的口形。有专门的程序可以帮你完成这些工作,在 CD 中提供了一些有关的链接。

B.4 小结

希望本附录所描述的体会和技巧对你有参考价值。创建游戏音频同其他事情一样,练习得越多,效果就越好。所以,我最后的一个提示是“不要放弃”! 多一点的耐心,一对好而严谨的耳朵可以帮助你创建令人满意的游戏音效。